NBSIR 81-2423

Compiler-Based Programming Support Capabilities

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Center for Programming Science and Technology
Systems and Software Technology Division
Washington, DC 20234

January 1982

Final Report

Contract NB79SBCA0131

Intermetrics, Inc.
4733 Bethesda Avenue
Bethesda, MD

NBSIR 81-2423

COMPILER-BASED PROGRAMMING SUPPORT CAPABILITIES

Gary Bray
Roger Lipsett
William Bail
Victor Berman

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Center for Programming Science and Technology
Systems and Software Technology Division
Washington, DC 20234

January 1982

Final Report

Contract NB79SBCA0131

Prepared for
Intermetrics, Inc.
4733 Bethesda Avenue
Bethesda, MD

U.S. DEPARTMENT OF COMMERCE, Malcolm Baldrige, *Secretary*
NATIONAL BUREAU OF STANDARDS, Ernest Ambler, *Director*

FOREWORD

The report that follows is the result of a contract effort initiated by the National Bureau of Standards´ Institute for Computer Sciences and Technology (NBS/ICST). This report was developed in connection with responsibilities under the Brooks Act (PL 89-306) which aims to aid Government agencies to improve cost effectiveness in the selection, acquisition, and utilization of automatic data processing resources. Efforts to satisfy these responsibilities include research in computer science and technology, and the development of Federal government-wide standards for data processing equipment, practices, and software. The software standards efforts comprise six families of standards, one of which deals with software quality control. Although it is recognized that software tools can aid in software quality control, NBS/ICST concluded that there did not exist a clear body of techniques for making effective use of tools. The present and related efforts are intended to fill this gap.

The purpose of this report is to record the results of an effort to determine a set of features offered by program analysis and testing tools that could be feasibly implemented in a compiler. Currently, program analysis and testing tools offer features that require syntactical analysis of a program in a manner similar to compilers. Much of the information that is generated during compilation could be used to aid program development in other ways. It is the goal of this effort to identify a set of software tool features and develop a methodology for combining these into a compiler.

Although the major goals of this effort are reflected in this report, a problem does remain. The report does not adequately cover what the cost impact of each of the features would be and what cost inter-relationships exist among features. For example, if one were to provide data flow analysis as a feature in a compiler, then this would make additional features such as program restructuring, program structure checking, complexity measurement, and certain types of program optimization much less costly to implement. In addition to the cost of implementation, there is a similar cost in compiler efficiency. For example, complexity measurement is much less costly in terms of compiler efficiency than data flow analysis. Because of limited funds for this purpose, these issues were not persued.

Raymond C. Houghton, Jr.
Systems and Software Technology Division

ABSTRACT

An effort to determine a set of features offered by program analysis and testing tools that could be feasibly implemented in a compiler is reported. Currently, program analysis and testing tools offer features that require syntactical analysis of a program in a manner similar to compilers. Much of the information that is generated during compilation could be used to aid program development in other ways. It was the goal of this effort to identify a set of software tool features and develop a methodology for combining these into a compiler.

Key Words: Compilers; Dynamic Analysis; Programming Aids; Software Development; Software Engineering; Software Tools; Static Analysis

The material contained herein is the viewpoint of the authors. Publication of this report does not necessarily constitute endorsement by the National Bureau of Standards. The material has been published in an effort to disseminate information and to promote the state-of-the-art of software development technology.

TABLE OF CONTENTS

1.0 INTRODUCTION

Software has become the most expensive component of computer systems. Every phase of the software life cycle is error-prone, and the investment of time and resources in software projects consistently has exceeded even pessimistic projections. When delivered, software systems frequently are unreliable long after installation and are difficult to use throughout their lifetime. These facts should not be interpreted as a condemnation of software developers; instead, they should be interpreted as a condemnation of the development environments and tools (or lack thereof) that allow, even encourage, the production of expensive, poor quality software.

High level language translators are major contributors in the effort to reduce the costs of software and to improve its quality and reliability. They relieve the programmer of many detailed, low level, and machine dependent considerations, making programming easier and programs more transportable. It is a well known result of empirical software studies that the number of debugged lines of code that an experienced programmer can produce per unit time is comparatively constant, regardless of whether the programming language used is an assembly or a high level one. Since the amount of work performed by a given number of high level language statements is several times larger than that performed by the same number of assembly language statements, high level languages are obviously more efficient in terms of labor time and, as a result, cost.

Different compilers for languages provide various degrees of assistance and support for their users. All presumably perform the essential function of translating the source program in a high level language into an equivalent target program in a low level language. Compilers with additional capabilities, however, can make programs easier to develop and debug. For example, a compiler that summarizes program characteristics or performs consistency checking can save considerable programming and debugging time.

In searching for ways of further improving software quality and lowering costs, researchers in academia, government, and industry have directed increased attention to the design and use of compiler-based tools to assist programmers in the generation of correct, reasonably efficient programs. As a result of this effort, the crucial role of programming languages and their compilers in the development of higher quality software is receiving wider recognition.

`s part of a broad effort to improve cost effectiveness in the selection, acquisition, and use of computer resources, both in the government and in the private sector, the Institute for Computer Sciences and Technology of the National Bureau of

Standards is investigating the feasibility and utility of incorporating programming support features into production compilers. This investigation is directed towards establishing standards for compilers used by agencies of the Federal Government and concerned private organizations. This report, intended to guide the development of compiler standards, analyzes the programming support capabilities that have been or may be included into compiler systems.

Most of the capabilities discussed are based upon the source program text or some representation derived from it. However, some capabilities cannot be exercised until the translated source program is executed with input data. Capabilities are associated with the typical phases, described in Section 2, occurring within modern compilers. This organization is intended to (1) place them in the context of the normal compiler activities, (2) suggest likely approaches to their implementation, and (3) facilitate understanding of their effects. This classification is not meant to be a hard and fast partitioning, but merely a useful mechanism within which capabilities can be structured and described.

The approach to programming reliability adopted in the past has often been to offset deficiencies in the programming language and implementation by developing separate, *ad hoc* tools, which perform checking or summarizing activities independently of the compiler. This approach has several undesirable consequences, both upon programmers and upon the performance of the computer system.

First, the program must be submitted to these tools for analysis either before or after compilation, introducing further development steps into the programming effort. The difficulty of managing the software development process unfortunately increases as the number of tools and steps involved increases. A support capability is more convenient, and therefore more likely to be used, if it is an option of compilation rather than a separate development step.

If support capabilities are contained within the compiler, any organization acquiring the compiler also acquires the support capabilities, promoting the transfer of technology among organizations. In this manner, programming tools become more widely available while reducing costs, since the expense of the support capabilities can be distributed among all the users of the compiler.

Another advantage of including support capabilities in a compiler is that since the compiler must perform a more in-depth analysis of the source program than a separate tool does, it has more information available for analysis and consideration. Also, the compiler has knowledge of the target language code that it generates, so that any insertions into the generated code required by support capabilities can be advantageously placed. Having a larger information base upon which to draw, the compiler has the opportunity to provide support more effectively and efficiently.

Moreover, the initial processing that these separate tools must perform often duplicates exactly the processing that the compiler must perform on the program at some point, introducing wasteful, redundant processing. If the capabilities are distributed among separate (and often incompatible or redundant) tools, significant efficiency is sacrificed, because the tools must be more general than if they were integrated into a single software unit. Considerable duplication of effort results when the program must be processed by the compiler and one or more tools that augment the compiler.

For example, one of the requirements of almost all tools is the reading and writing of the source program, a function that the compiler must also perform. Since input and output place considerable resource demands on present computer systems, the duplicated input and output processing results in inefficient usage of the computer system and, consequently, impairs program development. Another common activity that is frequently duplicated by tools and compilers is the recognition of identifiers, keywords, and operators in the source program. When a number of separate tools, all developed to provide specialized capabilities, must be used, the problems become more acute.

If the compiler performs the desired checking and support activities, the additional overhead, both human and machine, resulting from the use of separate tools can be eliminated. More recent languages (such as Pascal, Euclid, and Ada) have been designed so that their compilers are required to perform many of the support capabilities that previously have been performed by separate tools. The drawback of this approach is the additional complexity introduced into the compiler. The expense of the additional compiler complexity may be expected to be less than the costs of using and maintaining a host of separate tools, depending, of course, on the selection of a compatible, cohesive set of support capabilities to be included in the compiler. (Section 3 presents some guidelines for selecting such a set by means of a classification scheme.) Added compiler complexity is certainly less expensive in the long term than developing unreliable, error-prone software.

Chapter 1 presented background information on this report and provided reasons for including programming support capabilities in high level language compilers. Chapter 2 presents an overview of the organization of typical modern compilers and discusses the expanded viewpoint adopted in this report regarding compiler capabilities. Chapter 3 discusses the two classes of compiler-based capabilities: compile-time and run-time. Also, it presents an initial discussion of compiler capabilities that do not conveniently fall within one of the compiler phases outlined in Chapter 2. Chapter 4 discusses compile-time capabilities, organized according to the compiler boundaries outlined in Chapter 2. Chapter 5 discusses run-time capabilities, organized according to three categories: (1) capabilities dealing with control characteristics of the executing program, (2) those dealing with data aspects, and (3) those dealing with performance aspects. Chapter 6 ranks capabilities into one of three classes: (1) primary, (2) secondary, and (3) tertiary, according to their relative costs and benefits. Chapter 7 discusses some language characteristics and deficiencies of FORTRAN, Pascal, COBOL, and BASIC and suggests compiler capabilities that could be deployed to improve programming productivity in these languages.

2.0 COMPILATION ISSUES

The discussion of the compiler-based capabilities presumes a degree of familiarity with the organization and structure of typical compilers. This section presents an overview of the compilation process to establish the required terminology and points of reference for later discussion.

Also presented in this section is an expanded view of the role of a compiler in program development. This discussion provides a rationale for the inclusion of some capabilities into a compiler system that are usually thought to be beyond the realm of compiler implementations.

2.1 COMPILATION OVERVIEW

The control and data abstractions provided by high level languages are closer approximations than those of the low level machine to the way that people think about computations. Since few hardware machines can execute high level language programs directly, a translator is necessary to transform the source program into a language that a processor can execute. Translators may be broadly divided into two classes: compilers and interpreters. Compilers accept the source language program as input and produce an equivalent target language program as output. The target language program must then be executed in a separate operation. Interpreters also accept the source language program as input, but instead of producing an equivalent target language program as output, they execute each high level language statement as the statements are encountered. No equivalent target language program is produced. Although interpreters have important areas of application, compilers are more common and usually more efficient. This paper emphasizes the support capabilities that can be provided by compilers, although many of the capabilities are applicable to interpreters as well.

A compiler must have a number of basic capabilities to translate source language programs correctly. Broadly, these capabilities are lexical, syntactic, and semantic analysis, and code generation. Each of these capabilities is the basis of a distinguishable phase of modern compilers. Another capability, run-time support, is necessary to execute compiled programs correctly. The run-time support routines are closely interdependent with the compiler, although strictly speaking, they are not part of the compilation process. These activities are mentioned only briefly below to present common terminology and to provide context for the main subject at hand, compiler-based support tools.

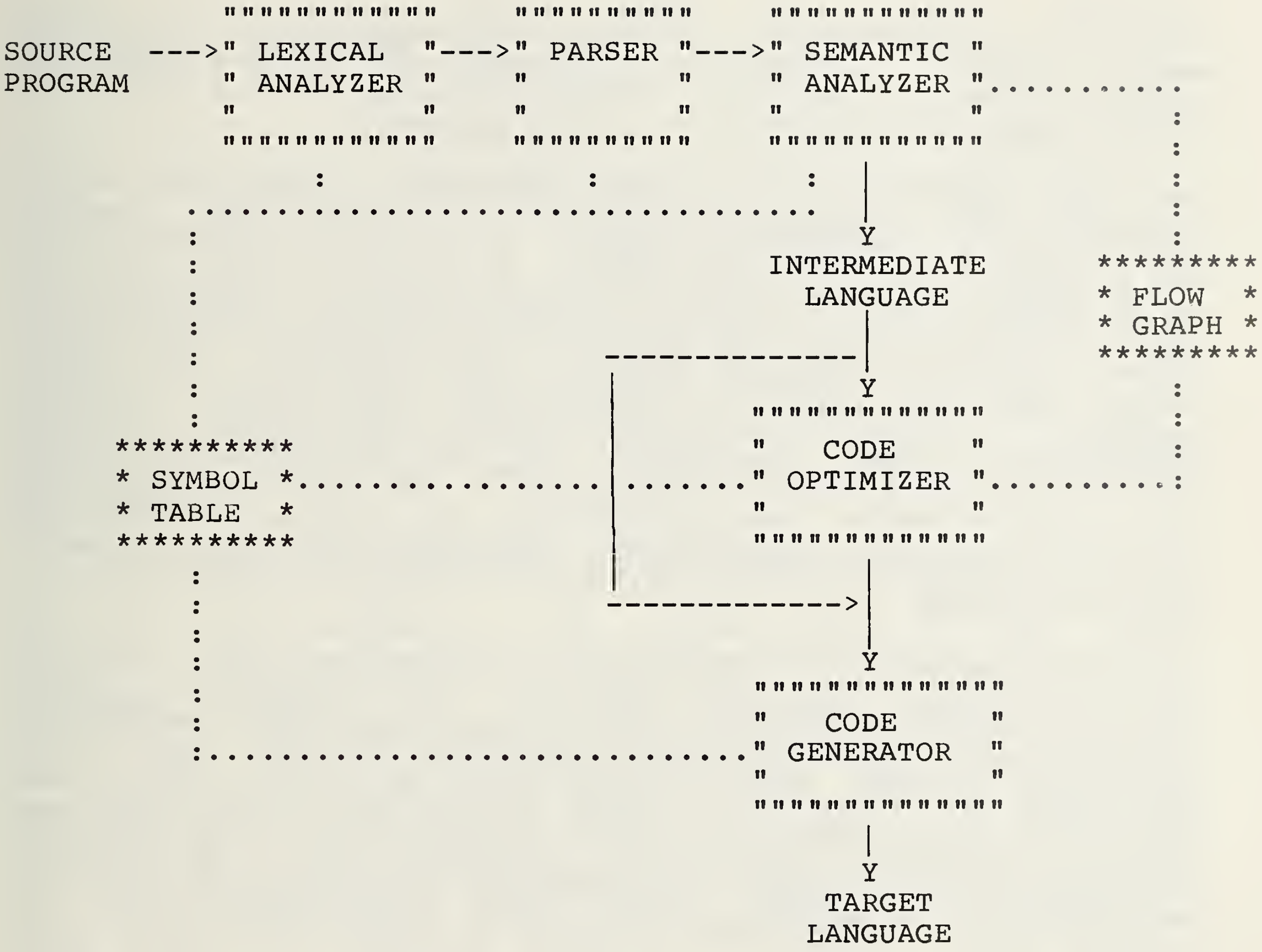

Figure 1. Organization of a typical compiler

Figure 1 presents a diagram of the organization of a typical compiler. Quotes (") enclose compiler phases and asterisks (*) enclose compiler data structures. The dashed lines in the Figure represent the pathways by which the source language program is transformed into the target language program, via the activity of compiler phases. The dotted lines indicate flow of information between phases of the compiler and principal compiler data structures.

A source program is a character string. A compiler must be able to recognize the primitive language units, or tokens, formed by this character string. A token is the basic element of a program, such as a numeric constant, an identifier, a literal character string, a reserved word, an operator, or a punctuation mark. The lexical analyzer partitions the source character string into language tokens, passing them sequentially to another compiler phase. Locating and diagnosing any lexical errors is inherent in lexical analysis. The program segment that performs lexical analysis is usually called the scanner.

The acceptable forms for object declarations and executable statements of a language are given by its syntax rules, usually specified by a context free grammar. A context free grammar is a formal set of rules that precisely specify the acceptable syntax for programs and, as a result, simplify compiler construction. Syntax analysis, or parsing, is the verification that the token sequence received from the scanner is consistent with these rules. Because parsing is the controlling phase of most compilers, syntax errors can pose difficulties during compilation. Often they result in the activation of some recovery procedure within the compiler so that the remainder of the source program can be processed. It is important that the translator be able to identify and accurately diagnose as many errors as possible during a single compilation to conserve user and machine time.

Syntax analysis typically directs another important activity of compilation: semantic analysis. The semantic analyzer verifies that the source program is correctly formed according to context sensitive, semantic rules of the language. Such rules govern, for example, the allowable uses of data objects and operators, the visibility or scope of identifiers, and permitted patterns of control flow. Any violations should be clearly diagnosed. The output of this phase of the compiler is commonly an intermediate language, which is a representation of the program that is closer to the target language (i.e., it is more primitive) than is the source program, but does not specify the detailed machine operations of the target language.

The scanner, parser, and semantic analyzer are target machine independent, i.e., they are concerned only with the correctness of the source program according to the language rules. Conceptually, these phases depend upon the characteristics of the source and intermediate languages, not upon the characteristics of the target language, although some implementations introduce target machine dependencies, in an attempt to improve the generated code. Frequently, one or more optional optimization phases follow semantic analysis, with the goal of improving the storage utilization and execution speed of the generated output from previous passes. Optimization phases can usually be divided into those that are target machine independent and those that are target machine dependent. The final phase of compilation is target language code generation. After code generation, the compiler is no longer directly involved in the development process; its influence is now embedded within the translated program. The final optimization phases and code generation are concerned with the details of the representation of the program and data in the target language.

Conceptually, these latter phases are source language independent, concerned only with the characteristics of the intermediate and target languages.

Several internal compiler data structures play central roles during compilation. The *symbol table* is the data structure that holds all information about the attributes of identifiers in the source program. An entry is made in the symbol table for each variable, type, procedure or function identifier encountered in the program during scanning and parsing. The symbol table entry for an identifier includes the name and a type description, containing information such as value constraints, bounds (if the identifier is an array), fields (if a record), and number and types of parameters (if a procedure or function). Routines in the various phases of the compiler consult the appropriate symbol table entry when they need to acquire information about an identifier.

If certain kinds of target language code optimizations are performed, the compiler often builds during compilation an abstract representation of the control flow of the source program. This abstract representation is called a *control flow graph*. A graph is a data structure consisting, in graph terminology, of *nodes* (or records) connected by arcs (or links) between the nodes. A node represents a statement or a block of statements. An arc connects two nodes and represents an execution pathway between the nodes. The arcs of the graph are *directed*, i.e., pathways are traversable in one direction only (corresponding to the fact that execution proceeds in one direction only). The compiler may, however, maintain backward links between nodes for convenient insertion and deletion of nodes.

The control flow graph encodes information about statements such as the number and locations of the predecessors and successors of each statement. Also, any information known about the values that program variables will have at a given point when the program executes can be attached to appropriate nodes of the graph. The analysis during compilation of values that variables will assume at given points when the program is executed is sometimes called *data flow analysis*. Both the symbol table and the control flow graph can be used by the compiler to provide programming support functions, as discussed in section 3.

The *run-time support environment* is closely related to the compiler. Run-time support consists of those activities that have been deferred, for reasons of feasibility or convenience, until the compiled program is executed. The run-time support needed varies with the language definition, but typically includes at a minimum procedures for input and output, and mathematical computations, such as trigonometric and exponential functions. Block structured languages and those with dynamically allocatable variables commonly require additional run-time support. The compiler and its run-time support are closely interrelated, as the preparations for invocation of run-time functions must be done by the compiler. The distinction between compile-time and run-time activities becomes unclear when considering interpreters, which have the capability to execute the source program rather than generating an equivalent target language program.

A compiler is more helpful if it detects errors at compile-time, when the source program is translated, rather than delaying error discovery until run-time, when the translated program is executed. Compile-time error detection saves wasted development steps on programs containing errors, conserving both user and machine time. Also, in many situations, a compiler can provide informative error diagnostics that would be more difficult to provide at run-time. If run-time error detection is unavoidable, an informative diagnostic should be issued by the run-time language environment, instead of relying on the system or hardware environment to handle the error.

2.2 AN EXPANDED VIEW OF COMPILER CAPABILITIES

Many suitable support capabilities discussed in following sections of this paper have traditionally been neglected in compiler implementations. One reason for the lack of support capabilities in most compilers is the influence of past hardware restrictions. Because of high hardware costs, compilers were designed to use the least possible amount of memory and CPU time. Hardware is now cheap enough and fast enough to support more comprehensive compiler capabilities.

Another factor influencing the lack of adequate programming support in many compilers is the youth of the computing discipline itself. Only recently have advances in programming methodology led to a clearer understanding of the kinds of capabilities that promote development of reliable, correct software. The techniques of current programming methodologies, usually collected under the term structured programming, are successful means of controlling the complexity of software and of promoting its reliability and maintainability. They include (1) top-down design, the development of programs by proceeding from the higher, more general levels of abstraction to the lower, more detailed levels, (2) information hiding, the shielding of data from routines that have no need to access it, and (3) data abstraction, the definition of data in terms of operations upon it, rather than in terms of representations of it.

The success in deploying these methods depends to a large extent upon adequate checking, enforcement, and support from the programming language and the surrounding programming environment.

Recent languages, such as Pascal and its descendants, provide language constructs supporting these methods. Moreover, these languages have been designed so that the programmer can make more information available to the compiler about the program and the problem it is intended to solve. Using this information provided by the programmer, the compiler can check the program for consistency and correctness.

But more comprehensive capabilities are needed to meet the demands of production programming environments. The capabilities offered by most current compilers have a lifetime of a single compilation. That is, the capabilities are stand-alone, depending upon no information other than that gathered during the present compilation. Little information (other than the target language program and the listing file) is maintained about a program after the compiler executes. The restriction to local, temporary information limits the kinds of capabilities that a compiler can provide.

For example, the ability to separately compile a program unit requires that the interfaces to other routines be checked for consistency of the number, types and modes of shared variables. This information is not available if the compiler relies solely on information gathered during the present compilation. Compilers that do provide separate compilation usually rely on the facilities of the underlying operating system environment, which are rarely simple or adequate. The central problem with relying on facilities provided by the system environment is that this environment is usually ill-suited for the compiler's purposes or needs.

The point of view taken in this paper is that the translation of high level language programs is but one capability of a broader language environment. The language environment is an integrated set of development tools that promote the construction of well-structured, reliable software in a particular programming language. Compilation is the central capability of such a language system; other capabilities of the language environment make the tasks of designing, coding, debugging, and maintenance easier. Thus, it is appropriate to consider the capabilities of the language environment to be compiler-based, although some of the capabilities are not completely embedded in the compiler, but depend upon or are part of the compiler's environment.

A compiler developed with programming support as a guiding goal cannot be separated from its surrounding language environment, which is designed specifically to interface simply with the compiler and to provide the kinds of functions necessary for the enhanced support capabilities of the compiler. The surrounding environment must be built upon the (often inadequate) facilities provided by the underlying system, but above this system interface the environment provides a set of functions tailored to the requirements of the compiler´s support capabilities. In addition to enabling capabilities that are not possible otherwise, including extensive support capabilities in the compiler reduces redundant processing and eliminates the need for a host of separate tools. This somewhat expanded usage of the term compiler includes some capabilities that are usually considered not to be compiler issues.

Some examples of capabilities usually not associated with a compiler are program construction, library management and symbolic debugging (see section 3 for a fuller discussion of each of these capabilities). A program constructor is an editor for a specific language, which allows construction only of syntactically correct programs. Library management is the maintenance of the information associated with a project in an integrated, automated manner. Symbolic debugging is the capability to interactively debug a running program in terms of source program constructs, such as symbolic names for variables and line numbers for control structures, rather than in terms of the machine language. These functions usually are not considered to be within the sphere of activity of the compiler and typically have been approximated in the past by some combination of manual procedures and *ad hoc* tools, or have been completely unavailable. According to the discussion of compiler systems given above, these capabilities are legitimate compiler concerns.

3.0 COMPILER CAPABILITIES

This section discusses the two major classes of capabilities, compile-time and run-time. Following the discussion of the two classes is an overview of capabilities that overlap the normal compiler phases. Several capabilities may be placed in more than one phase of the compiler, depending on the specific information that the capability is to provide. Often, for example, more than one phase makes contributions to summary information or measurements. The capabilities that are not necessarily local to a particular phase include cross referencing, auditing, profiling, complexity measurement, and library management. The general features of these capabilities will be discussed below in this section; the specific contributions made to these capabilities by the various compiler phases will be noted under the appropriate sections on the compiler phases.

3.1 TWO CLASSES OF CAPABILITIES

There are two major categories of capabilities: compile-time and run-time. These are discussed in Chapters 4 and 5 respectively. Compile-time capabilities are input data independent, i.e., they are based on an analysis of the source program text alone, without consideration of the actual input data that the program will process. At compile-time the input data is not available, which is both a limitation and strength of compile-time capabilities. The limitation is that the behavior of the program when run with particular input data on an actual machine, in general, cannot be determined. The strength is that information determinable from a compile-time analysis of the program is true regardless of the input data the program may be given to process on a particular execution. Thus, compile-time analysis is perforce incomplete, but the information that it does provide is invariant, holding for all executions of the program.

Run-time capabilities are input data dependent, i.e., they involve analysis of the program as it is running with a particular set of input data on an actual machine. Run-time analysis provides information about the characteristics of a program given a particular set of input data. This information, in general, will not be true of the program when executed with a different set of input data.

The use of the term run-time capability may seem somewhat unusual when applied to a compiler, as compilers are often regarded as having no direct role during execution. However, the compiler is involved in preparing information for use by run-time routines. Many support capabilities may be considered extensions or enhancements of the run-time environment. Just as the compiler must provide for invocation of run-time functions such as dynamic storage allocation, it can also provide for enhanced diagnostic and measurement functions. For example, in a compiler system that provides a symbolic dynamic debugger, one of the activities the compiler must perform is the preparation of symbol table entries in a format shared between the compiler and the run-time debugging environment. Although the compiler itself is not part of the run-time environment, symbolic dynamic debugging is not possible without the preparations made earlier by the compiler.

3.2 CAPABILITIES OVERLAPPING COMPILER PHASES

Information from several phases of the compiler is sometimes involved in providing a capability. Therefore, the capability cannot be implemented within a single phase of the compiler and must be spread among phases. A similar difficulty in localizing a capability to a single compiler phase results when a single capability name is applied to quite varied compiler activities. The compiler activities are similar in an abstract sense, but the detailed processing differs, depending on the focus of the capability. For example, the activities involved in developing a program profile vary with the kind of information that is to be included (possibly involving more than one compiler phase). The common features of some phase-overlapping capabilities, such as cross referencing, auditing, profiling, complexity measurement, and library management, are discussed in this section. The specific contributions of the particular compiler phases to these capabilities will be discussed under the appropriate subsection in the discussion of compile-time capabilities in section 4.

3.2.1 Cross Referencing

A cross reference is a list or summary of information about program elements, usually identifiers. A cross reference often summarizes information from lexical, syntactic, and semantic phases, organizing it according to source program features, such as routine name and source line number. The information deemed useful can be expected to vary somewhat with the characteristics of the source language. The cross reference should permit easy location of items of interest, helping indicate points in a program likely to be affected by proposed changes. The capability should support on-line retrieval of the cross reference information, possibly maintained by a library manager (see Sections 3.2.5 and 4.1.2).

3.2.2 Auditing

Program auditing is a general term for the examining of the source program for violations of standards, either language standards established by a standardizing agency or programming standards promoted by an organization. Language standards enhance the transportability of source programs by reducing the probability of incompabilities and unexpected errors when programs are moved from one installation to another. Programming standards are intended to promote the use of reliable, clear programming practices. Certain programming practices and constructs allowed by languages have been observed to account for a disproportionate number of errors. Further, the definition of some language constructs may be sketchy or incomplete, making the behavior of a program using them implementation dependent and thus not transportable. To avoid these problems, some organizations impose software standards, which prescribe the kinds of language and programming structures that may be used. These standards may pertain to lexical, syntactic, or semantic conventions. A compiler can monitor the source program for compliance to these standards, identifying and diagnosing any violations. Even in the absence of explicit standards, the translator may examine the source program for constructs which are suspicious, potentially dangerous or not transportable.

3.2.3 Profiling

Profiling is the collection of summary information, or the computation of statistics, about source program characteristics. A number of lexical, syntactic, and semantic features are readily summarized by a compiler. Profiles can be inspected manually or automatically for interesting or unexpected properties. Also, they can be used in empirical studies of language or software characteristics.

3.2.4 Complexity Measurement

Insofar as the application permits, complexity should be minimized in software systems. Complex programs or subprograms are more difficult to develop, test, and maintain; they are more likely to contain errors and more likely to conceal errors than simpler programs. A compiler can help identify complex subprograms by computing a metric which indicates relative complexity. The computed value may be used to assess software quality or to suggest program sections that should be redesigned or tested most extensively. Complexity metrics may include information from several program characteristics, based upon information gathered during more than one compiler phase.

3.2.5 Library Management

Library management is a term used to denote a wide collection of capabilities, spanning the typical phases of compilation. Some of these capabilities are textual, i.e., they simply manipulate collections of characters, without regard for any meanings the text might have when interpreted by other capabilities of the library management facility. Examples of textual capabilities that could be included in a library management facility are file comparisons, general text editing, and string searching. Other capabilities of the library facility are concerned with translation of library modules and the management of information related to translation. These capabilities are concerned with information that is meaningful in a particular language and directly support particular phases of the translator. Such capabilities include the expansion of shared modules in those routines that access the shared data, provision of global (inter-module) external cross references, and the maintenance of interface information so that programs may be compiled as multiple compilation units.

4.0 COMPILE-TIME CAPABILITIES

A compile-time capability is applicable to a program source in the absence of input specification. Such capabilities have often been called static or input data independent, although the distinction between static and dynamic, input data independent and input data dependent functions can be unclear, as in the case of interpreters or symbolic executors.

This chapter discusses the tool capabilities that would fit most naturally into the common compiler phases. To conveniently accomodate some of the support functions, two compiler "phases" are added to those mentioned in Chapter 2. A textual and a problem-semantic phase are added to the lexical, syntactic, and semantic (here distinguished as language-semantic) phases. The new phases are added to emphasize the nature of the capabilities included within them, although they probably will not be as distinct within the compiler as are the lexical, syntactic, and semantic phases. A textual capability such as macro processing may be a subphase of lexical analysis, whereas one such as library management may more accurately be considered as part of the compiler´s global environment. Problem-semantic capabilities are probably most conveniently implemented as subfunctions of semantic analysis.

In the discussion, the phases are ordered in the sequence of increasing information and knowledge of the compiler about the meaning of the source program. Capabilities are described within this sequence both to take advantage of the existing framework and terminology in describing their effects and to emphasize where in the compiler structure the tools may be implemented.

4.1 TEXTUAL

A textual capability views the program source strictly as a sequence of characters or text, not as a program. Such capabilities include file comparisons, text editing, and string searching. Note that capabilities concerned with language-oriented strings belong elsewhere in this classification.

4.1.1 Macro Processing

Macros consist of an identifier, called the macro name, and a character string, called the macro definition. The occurrence of the macro name in the source text is macro invocation and results in macro expansion, the replacement of the name by its definition. The definition becomes part of the source text, as if it had been entered originally instead of the name. Macros may be considered a short-hand or symbolic notation for a character string.

Macros can simplify and automate many of the tedious and error-prone activities of program development. Frequently, a part of the program text will be repeated in many places throughout the program. Using a macro processing capability, a name may be assigned to this text string and the name used wherever the text is desired. Macro calls ensure that the same text is substituted in all cases, eliminating possibilities of typing mistakes and other inconsistencies. They can be used to define symbolic constants, improve readability and make modifications easier.

A macro expansion capability provided by a compiler can be specialized or general. A specialized capability, which is simpler to implement, can perform only simple text substitution. Such a capability is useful to define symbolic constants in languages that do not provide for them or to define abbreviations for lengthy, often-used program text (e.g., FORTRAN common block specifications). A general, and more difficult to implement, macro capability provides more powerful facilities for text manipulation, including optional parameters, multiple levels of expansion, special counter and string variables for use in macro definitions, and operators for the special variables.

4.1.2 Library Management

Large software projects present major information management difficulties. Generally, several programmers cooperate on the development of large systems, requiring both the sharing of interface data and the hiding of implementation details. Current programming methodologies promote the development of many small routines in implementing large systems, requiring capabilities for separate compilation and system generation. Families of related system versions often tend to develop, requiring that distinct versions be kept separate while minimizing the amount of redundant information retained.

These requirements are usually met by conventions that are largely or completely manual. Many of these capabilities can be automated, however, as part of a language-based library management system. Translation is one capability of such a high level language system, which also provides integrated facilities for separate compilation with interface checking, system generation, documentation maintenance and generation, access control, and minimal encodings of different system versions. Data objects that are part of a development project are maintained on a data base managed by the library facility. Such objects may include source programs, target language programs, symbol tables, linkage information, documentation information, and system generation information. An integrated language system can simplify considerably programming and project management.

Library management is an important example of the expanded view, discussed in section 2.2, of the role of compiler systems in program development. The inclusion of a library management system into a compiler system is an attempt to automate some of the broader, more global activities involved in programming which have in the past been performed by manual practices or ad hoc tools.

The success in managing a large software project in the absence of integrated, automated systems usually depends upon management skills and the capabilities that happen to be provided by the system environment. A compiler system that incorporates library management facilities provides important features of its own environment, with minimal reliance on the capabilities of the underlying system. The components of the compiler system (one of which is the compiler itself) share uniform and simple representations of data objects (including source programs, intermediate language programs, and symbol tables) and observe standard, high level methods of accessing data objects. Closer control can be established over access to data objects, without being restricted to the methods provided by the host system.

Since the tendency is to view the library management facility as containing the compiler rather than the compiler containing the library management facility, it may seem more appropriate to consider compilation a library management-based capability. This view, however, does not emphasize that the library management capability is organized for the needs of the compiler. Library management fits as a compiler-based capability under the expanded view of compiler systems presented in Section 2.2.

Obviously, a library management facility supports many capabilities that are not textual, as discussed in Section 3.2. It has been included as a textual capability because the predominate flavor of its activities is textual.

4.2 LEXICAL

A lexical capability is based upon tokens, the primitive characters and character strings that are meaningful in a language. Thus, the data generated by a lexical capability are about identifiers, keywords, constants, operators and punctuation. Lexical capabilities are usually embedded in the scanner of the compiler.

4.2.1 Lexical Cross Referencing

A cross reference is a listing of occurrences of program elements, usually identifiers, organized in terms of some other feature of the program, such as the line numbers or modules in which those elements appear. The information contributed to a cross reference by the lexical analysis phase includes the names of all identifiers, constants, and operators encountered in the program, as well as the source program line numbers in which lexical items appear. Typically, the lexical analysis phase either enters this information into appropriate symbol, constant, and operator tables or passes the information to the parser for the parser to enter the information. In either case, it is a simple matter for the compiler to reproduce this information in a cross reference listing.

A cross reference should permit easy location of all lines, modules, or data areas in which an item of interest appears. This information may be used, for example, to trace identifier activity manually or to locate points in a program likely to be affected by proposed changes.

4.2.2 Report Generation

Generation and maintenance of documentation is a major aspect of software development. A translator can assist by providing the capability to extract comments selectively from source code. The extracted internal comments may then be used as the basis for external documentation or, if sufficiently complete, may be used without elaboration or modification. Such a capability can eliminate redundant effort and help maintain consistency between internal and external documentation.

A documentation extraction capability is simple to implement in a compiler, provided that documentation to be extracted is distinguishable from strictly internal comments, for example, by preceding comments intended for external documentation with a keyword. The scanner, upon recognizing the external documentation keyword in a comment, writes the remainder of the comment to a documentation file.

4.2.3 Lexical Profiling

A profile is a collection of summaries or statistics about source program characteristics. A number of lexical characteristics are readily summarized by a compiler. For example, the distribution of lexical items may be produced, giving a listing and count of each operator, operand, keyword, or other element in the program. A lexical profile can be used as a rough indication of the structure of a program, or it may be used in studies of software characteristics. A profile is easily implemented in a compiler by introducing counter variables or arrays for the characteristics to be profiled.

4.2.4 Lexical Auditing

Lexical auditing is a general term for the monitoring of a source program to detect lexical tokens that are, for some reason, unsafe or unwise. Some lexical elements, though not improperly formed according to lexical rules, can hinder transportability. For example, allowable numeric constants on one machine may exceed the precision of another machine. Other correct, though inadvisable, lexical constructs can impair readability. Examples of unwise lexical elements are FORTRAN identifiers and keywords that contain embedded insignificant blanks, making statements more difficult to read and understand. Also, some implementations of languages relax certain lexical restrictions, allowing, for example, increased identifier lengths and inclusion of special separator characters in identifiers. A compiler performing lexical auditing can detect standards violations and suspicious lexical constructs.

Other standards may govern documentation conventions within an organization. Documentation standards dictate the amount and placement of comment statements in the source program. For example, it may be required that procedure headings be followed by a comment preamble, explaining the actions of the procedure. The lexical phase, armed with information from the parser indicating that a procedure has been entered, can check for compliance with this standard.

Lexical auditing is, in general, simple for the compiler to perform. It usually involves restricting somewhat the class of lexical tokens that the scanner recognizes as acceptable, and thus introduces no new major data structures or operations.

4.2.5 Lexical Complexity Measurement

A simple complexity measure may be derived from lexical information alone. Some function of lexical characteristics may be developed, empirically or theoretically, for a given language to indicate the lexical complexity of a program in that language (Halstead, 1977). The scanner maintains a set of variables representing the lexical characteristics of interest, such as the number of operators, operands, and other tokens in the program. When lexical analysis is complete, the compiler computes the lexical complexity by inserting these variables into a predefined complexity function for the language. Although simple to compute, the lexical complexity measure may not closely correspond to intuitive notions of program complexity and, in such cases, it may be desirable to combine lexical information with syntactic and semantic characteristics to form a set of complexity measures.

4.3 SYNTACTIC

A SYNTACTIC capability is concerned with all characteristics of the program that are associated with parsing. This area has been very well studied over the years and therefore has many tools to support it.

4.3.1 Program Formatting

One of the clearest ways to convey a program´s structure is by means of statement indentation. If a program is properly indented, a reader can grasp at a glance the statements which are subordinate to and controlled by other statements. Conversely, inappropriate indentation can mislead a reader, suggesting nonexistent relations among statements.

Proper statement indentation, called program formatting or pretty-printing, can be done by the language translator. As compilation progresses, the compiler computes and maintains information about the proper layout of the source text, formatting the listing accordingly. In addition to formatting the listing, the compiler may have the capability to rewrite the source file appropriately indented. Program formatting is applicable particularly to block structured languages, but is useful in any language if programming or structuring conventions have been established.

4.3.2 Program Constructing

The translator organization described in Section 2.1 may be termed an analytic approach. The input representation of the source program is a character string that must be analyzed for lexical and syntactic correctness. Typically, an editor that operates on character strings is used to create the program. The editor is unaware of the intended use of the character string as a source program and, thus, makes no distinction between legal and illegal character strings in a given language. A significant portion of the translator´s efforts, therefore, must be devoted to verifying the correctness of these strings.

It is not obvious that character strings are the best program representation to be shared between the creation and translation phases of program development. An alternative approach is to employ an editor which is designed specifically for program development in a particular language and produces as output not a character string that needs to be analyzed, but an intermediate language suitable for immediate translation. Such an editor, called a syntax-directed editor, or program constructor (Huet, et. al., 1977, Teitelbaum, 1979, Habermann, 1980), operates upon syntactic constructs rather than upon lines and characters. Language keywords and syntactic forms are not typed by the user; instead, they are inserted by the program constructor in response to user commands. A command is provided for each syntactic construct in the language. Typing a constructor command places a template for the corresponding syntactic primitive at the current editing position, which may then be expanded and elaborated by the user.

To illustrate, instead of typing the characters for an IF-THEN-ELSE statement, a user types a constructor command, for example .IE, which places the tokens comprising the statement, appropriately formatted, at the current editing position. That is, the .IE command displays the text

```
if <condition> then
   <statement>
else
   <statement>
```

at the current editing position, if syntactically correct at this point. Nonterminals, the symbols enclosed in < ... > above, for the conditional expression and statement that constitute the IF statement are displayed, pending specification by the user. A nonterminal is a symbol that is not part of a legal program, but rather represents or stands for a class of language constructs, which may be substituted in the place of the nonterminal. Eventually, all nonterminals must be replaced by identifiers and operators before the program is submitted to a translator.

A number of advantages result from use of a program constructor. Most important, the constructor is responsible for insertion of syntactic items, not the programmer. Thus, it is unnecessary to parse programs, and problems of misspelled or mismatched keywords and incorrect punctuation are eliminated. It is impossible to make syntax errors.

Moreover, the constructor may be expected to operate upon some intermediate representation of the program, such as an abstract syntax tree, instead of character strings. Using an intermediate form, the editor is not burdened with unnecessary string manipulation. This intermediate representation can be used directly as input to a translator to produce target language instructions. Thus, the translator can be much simpler than typical compilers (at the expense, of course, of the more sophisticated editor), as its input is an intermediate form that need not be parsed. A separate pretty printer is not needed, as the constructor can produce indented, structured text representations of the program.

4.3.3 Error Correction

When using many compilers, fatal compilation errors often result from trivial syntactic errors in the source program. It is not uncommon for a compiler to issue a fatal diagnostic such as "semi-colon expected" as a result of a minor oversight by the programmer. It is not unreasonable to ask that the compiler assume the expected semi-colon, if one is necessary at a point during compilation.

Rather than simply issuing a diagnostic when errors are encountered during compilation, it is possible for the compiler to attempt a correction. The correction is a guess, possibly based upon some heuristic or cost function, about what the programmer intended. The correction usually involves inserting and deleting lexical tokens until a syntactically-correct program results. In addition to generating target language according to the attempted corrections, the compiler writes a new, modified version of the source program, containing the annotated corrections.

For example, in a language that separates or terminates statements with semi-colons, errors of omission or commission involving semi-colons have a high probability. An appropriate heuristic or cost function for such a language makes insertion and deletion of semi-colons high-probability corrections. Most correction schemes attempt to retain as much of the original source text as possible, so the compiler usually emphasizes generating tokens to insert into the program.

Clearly, some errors will be too severe for the corrected program to be useful, but if the correction scheme is well suited to the language, many errors can be accurately corrected. Some ad hoc and formal (Fischer, et. al., 1977) correction techniques, most based upon parsing, have been developed that result in appropriate corrections for many kinds of errors. Error correction performed by the compiler is not a substitute for clear and careful programming, but it can reduce the number of compilations needed during software production and debugging.

4.3.4 Syntactic Profiling

A syntactic profile of the source program may be produced by the compiler. Such a profile may include a count of each kind of statement, the nesting level of compound statements, and the depth of loop nestings. This information is easily gathered by the compiler during parsing by maintaining counters of statement types and nesting levels. Such a profile can be useful to convey rough estimates of program complexity and performance, and may be useful in empirical studies of the characteristics of programs in an organization.

4.3.5 Syntactic Auditing

Syntactic auditing by a compiler can detect syntactic constructs that violate software standards that have been established by an organization. Standards may govern syntactic characteristics such as (1) the nesting depth of loop statements, important in time-critical applications in which deeply nested statements can cost considerable execution time, (2) the number of source module statements, important to the efficiency of a virtual memory paging system and to software simplicity and maintainability, and (3) the permitted types of control flow statements, also important to simplicity and maintainability. For example, a software project may restrict the nesting of loops to three levels to prevent excessive execution time being spent in highly nested code. Also, time-critical applications may restrict the size of target language modules to some small number of virtual memory pages to reduce paging overhead. To enhance understandability, an organization may restrict or forbid the use of GOTO statements or alternate returns from subprograms. As with lexical auditing, checking these program characteristics in a compiler is inexpensive, usually involving restricting the class of syntactic constructions that are acceptable. Counter variables and modified syntax rules suffice to implement most syntactic auditing capabilities.

4.3.6 Syntactic Complexity Measurement

Syntactic information may be used in deriving complexity metrics. The kinds of syntactic information that can be used as variables in a complexity measure include the number of syntactic constructs, the types of syntactic constructs, the nesting depth of statements, the number of predicates, and the number of operands in expressions. Some function of these factors can be derived for a given language to characterize the syntactic complexity of a program.

One possible measure to characterize control flow complexity is based upon the cyclomatic number of a graph (McCabe, 1976). The cyclomatic measure is a simple function involving the number of nodes and arcs in the program control flow graph and correlates closely with intuitive understandings of control flow complexity. Although it is necessary actually to construct during semantic analysis the control flow graph for programs using unrestricted transfers of control (see Section 4.4.3), the cyclomatic complexity measure for a structured program, i.e., a program using only single-entry, single-exit control structures, such as WHILE, REPEAT, and IF-THEN-ELSE statements, is computable from syntax information alone. It is simply the number of conditions, or primitive predicates, in the program plus one. The compiler can maintain a count of conditions during parsing, and if the program uses only structured statements, the compiler easily can compute the cyclomatic complexity measure. If the program is not well-structured, syntactic information alone is insufficient to compute the cyclomatic number, and the control flow graph must be constructed (see Section 4.4.3 on semantic complexity measurement below).

4.4 LANGUAGE-SEMANTIC

A LANGUAGE-SEMANTIC capability is concerned with how the language constructs are mapped into their meanings. The domain of these tools covers the context-sensitive characteristics of the language. Capabilities falling into this category analyze a program with regard only to language semantics, without consideration for the problem the program is intended to solve.

4.4.1 Semantic Cross Referencing

The semantic phase of compilation can add helpful information to a cross reference listing. This phase can list important attributes of identifiers, such as their types, storage allocation, bounds, initial values, and value constraints. Other important information that can be produced in a cross reference with contributions from the semantic phase includes a list of statements that reference a variable, a list of statements that modify a variable, and a list of variables that are external to a module.

A compiler can provide summary information about procedure and function calls. A helpful representation of program structure is a procedure call hierarchy. The procedure calling pattern may be represented as a tree, a node representing a procedure, and its children representing the procedures it calls. A compiler can produce some printable representation of the tree of procedure calls in a program as part of a cross reference capability.

A cross referencing capability in a compiler is simple and inexpensive to implement. Since the compiler necessarily has all the information about identifiers available in the symbol table, the only additional effort required to provide a cross reference listing is formatting and printing this information.

4.4.2 Semantic Profiling

A program profile can be more revealing if it contains information about the semantic characteristics of the source program. The semantic phase of a compiler can contribute such information as the number of intrinsic and user-defined types in the source program, the number of each parameter passing mode used in the program, and the number of type coercions performed. As with lexical and syntactic profiling, a semantic profile is easily implemented by counter variables within the compiler.

4.4.3 Semantic Complexity Measurement

Complexity measures can be derived during semantic analysis that are more sophisticated than those based solely on lexical and syntactic information. The cyclomatic number of a graph (see Section 4.3.6) is a useful measure of program complexity. Although syntactic information is sufficient to compute the cyclomatic number for a structured program, a program using unstructured control flow requires information from the semantic phase of compilation. To compute the cyclomatic number of an unstructured program, the compiler can construct a control flow graph, the nodes of which represent statements and the edges (or arcs) of which represent transfers of control. The complexity of the program can be estimated from properties of the control flow graph, such as the number of nodes and edges, possibly supplemented by information about lexical and syntactic properties.

4.4.4 Flowchart Generation

A flowchart is a diagrammatic representation of an algorithm. It can be a useful tool in understanding the structure and control flow of a program, particularly in languages lacking structured control statements.

In the course of translation, a compiler can automatically produce a flowchart for the source program. In addition to helping a programmer better understand an algorithm, it can be used as part of system documentation and as an aid in maintenance.

4.4.5 Interface Analysis

Any variable that is not declared, implicitly or explicitly, to be local to a routine is by definition shared by that routine with its environment. A shared variable is made accessible to a routine by including the variable in the parameter list of the call to the routine or by declaring it in a global or shared data area visible to the routine.

Interface errors are among the most common in programming, particularly in larger systems on which several programmers cooperate and exchange data. Inconsistencies in the number, types, modes, ordering, and storage allocation of shared variables can be very costly and should be detected at an early stage of development, preferably compile-time. Separate compilation is the ability to compile a program as a number of related compilation units, rather than as a single monolithic unit, with interface checking. Multiple compilations without interface checking is called independent compilation.

Languages that are strongly typed and emphasize compile-time checking require the compiler to check interface specifications. Compilers for languages that are not strongly typed can, nevertheless, check interface consistency. For such languages, summaries of interface anomalies can be generated, or if programming standards concerning interface consistency have been established, standards violations or error conditions can be reported.

If the language design requires that routines constituting a program be compiled as a single unit, interface checking is as simple as ordinary type checking. All necessary information about data objects, including those shared between routines, is specified in the compilation unit.

If procedures may be separately compiled, maintaining interface integrity is more difficult. The compiler must maintain information about interfaces between compilations and be able to access this information when a separate routine is to be compiled. Typically, the information is stored on a data base of the host system and made available to the translator at the start of compilation of a related unit.

4.4.6 Semantic Auditing

The discovery of many poor programming practices, which often are violations of software standards imposed by organizations, can occur during the semantic analysis of the source program. This section mentions several examples of useful semantic auditing capabilities.

Implicit variable declarations and implicit typing rules in a language often cause errors. If the explicit declaration of a variable is omitted, the variable may be given a type other than the one the programmer intended, resulting in inaccuracies when arithmetic and comparisons are performed with the variable. Such problems can be avoided by requiring the mild inconvenience of explicit declarations of all variables. When using languages that allow implicit variable declarations, organizations often adopt the convention that all variables must be explicitly declared. Semantic auditing can ensure that this convention is obeyed. The compiler can simply check that program variables used in executable statements already have declaration entries in the symbol table, complete with type information.

In some languages, the use of a literal constant as an actual parameter, i.e., as a parameter in a call to a subroutine, is an unsafe practice, depending on the mechanism for parameter transmission adopted by the compiler. If the mechanism is always to pass the address of parameters to subroutines, commonly the case in FORTRAN, then passing a literal constant to a subroutine can change the value of the literal constant. This difficulty occurs if the subroutine assigns a new value to the formal parameter, the variable listed in the subprogram heading, corresponding to the literal constant actual parameter. Although some compilers may handle this problem in a manner that avoids changing the value of the constant, the program is not transportable to other compilers, because many handle the situation erroneously.

To illustrate, consider the following FORTRAN subprogram and example call:

```
subroutine increm(num)
integer num
num = num + 1
return
end

integer four
call increm(2)
four = 2 + 2
print *, four
```

When run with many FORTRAN implementations, this fragment would print the number 6, because the subroutine would have changed the value of the literal constant 2. Although the error is obvious in this contrived example, the same situation is seldom so obvious in actual programs.

A simple capability to help avoid this problem is for the compiler to check subroutine calls for the use of literal constants as parameters and warn of possible errors, regardless of whether, in fact, an assignment is made to the corresponding formal parameter. Having been warned by the compiler that a literal constant is passed to a subroutine, the programmer can check the subroutine to ensure that no assignment is made to the corresponding formal parameter (which can easily be determined if the compiler summarizes in a cross reference listing the statements that assign to a variable). A preferable solution to this problem is an auditing capability that warns of errors only in the case that the subroutine actually performs an assignment to the corresponding formal parameter, eliminating the requirement that the programmer perform the check manually. This solution is more difficult, requiring interface analysis (see section 4.4.3) to determine the actual parameters that can be changed by a subprogram. Part of the information about the interface to a routine is the parameters that can be given new values by the routine. If a constant is passed in the position of a value-receiving parameter, the compiler diagnoses the subroutine call as unsafe. If the parameter cannot receive a new value from the subroutine, the compiler issues no warning.

Sometimes syntactically permissible constructs have incomplete semantic definitions. Examples are FORTRAN 66 DO statements and computed GOTO statements. The actions of these statements for several frequently occurring situations is undefined, possibly making programs using them implementation dependent. Since they are major control constructs in FORTRAN, these statements can hardly be prohibited by software standards, so syntactic auditing is inappropriate. A compiler can generate code to perform a run-time range check of the value (see section 4.4.5), but this solution does nothing to enhance the transportability of programs.

A more transportable approach to incompletely-defined statements is to adopt a standard requiring an explicit check in the source code that values are appropriate before executing a particular construct. For example, a check that the index value is within the defined range for a computed GOTO in FORTRAN may be necessary to avoid implementation dependencies. A compiler can ensure that such a check is performed for constructs having incomplete definitions. To provide this capability without arbitrary restrictions on the placement of the checks, data flow analysis may be required to ensure, for example, that the value of a GOTO index has been checked on all execution paths leading to the GOTO.

If transportability is of sufficient priority, another implementation possibility is to modify slightly the syntax rules, or context-free grammar, of the language that the compiler accepts. The grammar rules are changed so that an ill-defined statement must be preceded by a check for appropriate values. For example, the CASE statement in Pascal is undefined if the value of the selector variable is not one of the selector values enumerated in the CASE statement. The grammar recognized by the compiler may be changed so that CASE statements must be preceded by an IF statement. A semantic check ensures that the condition tested in the IF guards against the selector variable not having a legal value.

4.4.7 Range Checking

Some languages, such as Pascal and Ada, allow range constraints to be specified on variables. Constraints are upper and lower bounds on variable values. They may be considered global assertions about the value of variables. To illustrate, the declaration

```
     index:  0..5;
```

defines INDEX to be a variable, such that 0 <= INDEX <= 5. In making such a declaration, the programmer states that only these values are meaningful for INDEX, and any attempt to give it a value outside of this range is meaningless and, therefore, is an exception condition. The compiler must verify that the values for INDEX never fall outside its acceptable range. To do so usually requires target language code to check the value in statements assigning to INDEX. Range constraints improve readability, promote security, and reduce debugging effort. They may also affect the efficiency of the generated code or the size of the data area.

Regardless of whether the language provides a range constraint capability, some language constructs implicitly define range constraints on values. For example, a static array declaration defines the allowable values that may be used as indices. Similarly, case statements and indexed jump statements implicitly specify the meaningful values of the index expression. The compiler should provide for verification that acceptable values are used in these constructs.

Range checking is particularly appropriate when performed on array subscripts, as errant array indices are a common problem. A range checking compiler can possibly detect some out of bounds array references at compile-time. In other cases, it must produce target language statements to perform the check at run-time. If an unacceptable value is encountered, a diagnostic may be issued, an exception raised, or the program aborted. Regardless of when it is done, the compiler should provide for detection of array indices that are out of bounds, rather than having the program behave mysteriously or develop other errors as a side effect. These comments are equally applicable to case statements and indexed jumps.

If the language permits value constraints on variables, the compiler must check all uses of the variable for acceptable values. If a language does not permit range constraint specification, a compiler can, nonetheless, perform some checking. In particular, checking of array subscripts and of index variables in indexed jumps should be compiler options. To provide a more general range checking capability in languages that do not otherwise provide for them, range constraints can be specified in comments or compiler pragmas. A pragma is an instruction to the compiler that controls subsequent compilation. A pragma specifying a range constraint instructs the compiler to check that values assigned to the variable are within the specified range.

Range checking is inexpensive compared with its benefits. Often the compiler will in all cases generate target language code to perform a check for an acceptable value. If the compiler performs data flow analysis, it may be able in some cases to verify that range constraints are satisfied by compile-time analysis. In such cases, no checking code need be generated.

For example, consider the following declarations and statements:

```
intvar:  integer;

smallvar:  0..5;

intvar:= 5;

smallvar:= intvar;
```

A compiler that performs data flow analysis is able to determine that the assignment of intvar to smallvar is always permissible, since at the point of the assignment the range constraints of smallvar will always be satisfied.

4.4.8 Reference Analysis

Common errors in referring to variables are the use in an expression of a variable that has not been given a value along some execution path and the assignment of a value to a variable that subsequently is unused along all execution paths. The former anomaly is likely either to result in an exception condition at run-time, or worse, to produce erroneous results if the path is executed. The latter is wasted computation and may indicate a more serious error, such as the omission of one or more statements that use the variable. Using a control flow graph, reference anomalies of these two types can be detected at compile-time (Osterweil and Fosdick, 1976); such detection is called reference analysis.

In addition to revealing omissions of variable initializations and uses, reference analysis can detect another type of error. If the language allows implicit variable declarations, reference analysis will diagnose a misspelled identifier to be either an uninitialized variable, when found in an expression, or an unused variable, when the target of an assignment statement.

To detect reference anomalies, the compiler performs data flow analysis using the control flow graph constructed from the source program. Information about whether a statement assigns a value to a variable, uses the value of a variable, or does both is associated with the statement nodes. Using this information attached to the control flow graph, a compiler can detect reference anomalies at compile-time, eliminating many occasions of run-time debugging and abort diagnosis and possibly preventing critical errors after system installation.

4.4.9 Program Restructuring

Program restructuring is an active form of program auditing. Whereas auditing is the detection of standards violations or undesirable practices, restructuring is the modification of the source program to eliminate deviant code. A compiler which performs program restructuring detects constructs which violate software structuring standards and replaces them by equivalent, approved ones. For example, a program using unstructured control flow may be altered to use only well-structured, single-entry, single-exit control statements. The compiler generates a new source file containing the restructured program, as well as generating the target language program.

A restructuring capability is primarily useful in upgrading older, unstructured programs that, for some reason, are infeasible to rewrite, but that, nevertheless, must be maintained. An organization having this need would find a restructuring compiler helpful. For example, an organization with a large investment in FORTRAN 66 programs may find them more maintainable if they are compiled using a FORTRAN 77 compiler that performs restructuring on FORTRAN 66 programs. Such a compiler would, for example, replace logical IF statements and interleaved GOTOs with equivalent IF-THEN-ELSE statements. Restructuring should not, in general, be used as a substitute for careful, well-structured programming.

Program restructuring requires that the compiler construct a control flow graph of the source program. The compiler searches the control flow graph for control flow patterns that correspond to those resulting from structured control statements. Upon finding a match, the unstructured source program statements responsible for the matched control pattern are replaced by the equivalent structured control statement. A restructuring compiler probably will be unable to replace all unstructured statements, as some patterns of unstructured control flow are irreducible to structured ones without the introduction of new variables.

4.4.10 Program Structure Checking

Flaws in program structure are usually undetected by ordinary syntactic and semantic analysis. Structural errors include redundant statements, unreachable statements, statements without successors, and unreferenced labels. Although flaws of this nature do not violate syntactic or semantic rules, they do not add information to the program and may indicate more serious logical errors. Identifying them at compile-time can avoid later debugging costs.

A compiler can discover structural flaws by constructing a control flow graph. Nodes of the graph represent statements (or sequences of statements) and the edges between nodes represent the flow of control. Traversing this data structure can reveal structural anomalies in the source program. (The control flow graph may prove useful to other compiler capabilities; for example, see the section on complexity measurement.)

4.4.11 Type Analysis

Variable declarations associate a data type with an identifier. A data type expresses properties about objects of the type and governs the operations which may be meaningfully performed on them. Types are useful both to the language user, to define the abstract properties of variables, and to the translator, to make internal decisions concerning storage allocation and selection of target language instructions. Consistent use of variables according to their types enhances the simplicity, clarity, and maintainability of programs.

Type analysis is a mandatory activity in translators for some languages. Variables in strongly typed languages have a single type. Each type in the language has a set of operations permissible for that type (e.g., arithmetic for integers, concatenation for strings, and logical operations for booleans). Variables of one type cannot be used in operations which require or involve objects of different types. Such languages require that the compiler check the consistency and compatibility of operations on data objects, permitting no automatic type conversions, known as type coercions.

For example, if CH is a character variable, it is not permitted to use CH in a multiplication operation. The compiler does not automatically convert (coerce) the value of CH into an integer value. Any conversions between types is done explicitly by use of predefined type conversion functions. Explicit function calls ensure that the programmer intends a type conversion and considers it meaningful in this context. For example, if a programmer indeed needs to use the value of CH in a multiplication, it is necessary that the variable be explicitly converted by prefixing it with a type conversion function, as in INTEGER(CH). It is widely accepted that this design approach promotes reliability, security, clarity and readability. Errors may be found at compile-time that might otherwise require extensive dynamic debugging or remain undetected long after system installation.

If a language does not strongly type data objects, the compiler should issue warnings or diagnostics when a variable is used inconsistently or an automatic type conversion is performed. Type analysis is particularly useful in languages which allow implicit variable declarations and automatic type coercions. For example, omitting a declaration of a variable in FORTRAN can, through implicit declarations and conversions, result in a program which compiles without errors, but which produces unexpected results. Type analysis of such a program would warn the user of type conversions, alerting the programmer to an erroneous implicit type declaration. For example, if the declaration of a variable intended to be real is omitted, and if the variable is implicitly typed integer, the subsequent assignment of a real value to the integer variable will be noted as a type violation, avoiding the possible loss of precision.

4.5 PROBLEM-SEMANTIC

A PROBLEM-SEMANTIC capability is concerned with how the problem statement is mapped into the program, and thus deals with verifying that the program computes the desired function. The problem statement can be formal, or exist only in the mind of the programmer. Such tools have as their domain not only the context-sensitive characteristics of the program, but also the problem statement.

The ordering of the problem-semantic capabilities below is intended to provide a rough ranking of the capabilities according to the difficulty and complexity of their implementations.

Capabilities that are easier to implement are presented before those that are more difficult to implement.

4.5.1 Units Analysis

Related to the topic of type analysis is units analysis. Variables may be given a unit attribute as well as a type. The unit attribute specifies that a variable is always associated with and measured in the given unit. Since most languages currently provide no notation for them, units specifications may be included in comments or pragmas.

A compiler can check units consistency using algebraic laws of composition and cancellation. If the unit of a variable is incompatible with that of an expression being assigned to the variable, a units error has occurred, indicating an incorrectly formed expression, an erroneous formula, or missing terms. Units analysis is most beneficial in scientific applications.

For example, using a compiler that performs units analysis, a programmer may declare a variable to have the attribute of being measured in cubic meters. A statement that assigns a value to the variable is checked by the compiler to confirm that the value also has the attribute of being measured in cubic meters. To illustrate, in the following declarations:

```
length, width, depth: real meter;
volume: real meter * meter * meter;
```

the variables length, width, and depth are declared to be of real type, measured in meter units, and the variable volume is declared to be also of real type, but measured in cubic meter units. Using this information, a compiler can determine that the first assignment below is correct and the second incorrect:

```
volume:= length * width * depth;
volume:= length * depth;
```

Note that ordinary type analysis would be unable to detect the incorrect assignment to volume.

Units analysis depends on the ability of the compiler to perform limited algebraic manipulations on symbolic expressions. The units of the terms of an expression are composed or cancelled, as appropriate, yielding a unit attribute for the final value. If the unit attribute of the expression is not the same as the unit attribute of the target variable of the assignment, the compiler flags the statement as containing a unit error.

4.5.2 Assertion Checking

Assertions are statements about variables or relationships among variables, usually expressed in terms of boolean conditions. They specify the variable values or conditions that the program expects and can meaningfully process at that point. In addition to improving documentation and readability, assertions can improve the reliability and security of programs when checked automatically.

Assertions may be global or local in scope and generally can be placed anywhere in the program that an executable statement can appear. Global assertions specify conditions which are expected to prevail from the point at which they are encountered until execution of the source program ends. Local assertion statements are expected to be true only where they are encountered during execution.

Compilers can provide for checking assertion statements either at compile-time if possible or at run-time if necessary. Using data flow analysis, a translator may be able to verify an assertion statement at compile-time. For example, assuming a compiler that checks the condition following an ASSERT keyword, the following assertion need not be checked at run-time, if the compiler performs data flow analysis:

```
saveindex := 0;
index:= 1;
while index <= 10 loop
   if info[index] = desiredvalue then
      saveindex:= index;
   else
      index:= index + 1;
   end if;
end loop;
ASSERT saveindex < 11;
```

Data flow analysis enables the compiler to propagate the value and constraints of the variable index, so that the value of saveindex is known to satisfy the assertion condition at compile-time.

If the compiler is unable to determine the truth of an assertion at compile-time, it can add target language statements to check these assertions at run-time. Section 5.2.3 discusses the run-time aspects of assertion checking.

4.5.3 Symbolic Execution

Rather than verifying correct program behavior through testing with a small set of test input values, programs may be verified for classes of input using symbolic execution (Boyer, et. al., 1975, King, 1975, King, 1976, Clarke, 1976, Howden, 1977). Symbolic execution is the use of symbols to represent a fixed data value. The input variables to a program or procedure are given symbolic values. These symbolic values are manipulated as a result of program assignment statements and control flow, using algebraic and boolean principles to simplify expressions as symbolic execution proceeds. When symbolic execution terminates, the symbolic expressions for the output variables may be examined to verify that they are the correct function of the input symbols.

Because variables are represented symbolically rather than by specific data values, it is usually impossible to determine which branch of a decision statement will be taken during symbolic execution. Therefore, some method of path selection must be provided to specify the execution paths that are of interest. Selection may be provided by assumptions about ranges of data values or through explicit specification of branches to take. To provide path control, the symbolic execution system may have an interactive command language.

A symbolic execution capability would substantially complicate a compiler, if current techniques for symbolic expression manipulation are used. Presently, most symbolic execution systems only approximate the functions, generality, and efficiency that would make them practical development tools.

4.5.4 Test Data Generation

The primary practical method of ensuring program correctness is testing. Testing is the execution of the program with data for which the correct results are known or are easily determined. The observed results are compared with the expected results, and if they are consistent, the program is confirmed to be correct for that data. Since the number of possible test cases is usually very large or infinite, exhaustive testing is infeasible, making test data selection a crucial aspect of software testing.

Several criteria for an effective set of test cases have been proposed. One criterion, called statement testing, is that all statements in the program are to be executed at least once. Executing every statement does not necessarily involve traversing all possible execution paths, which are determined by the branching points in the program. Branching points are determined by the control flow logic of the program and may be either explicit, in the case of unstructured GOTO statements, or implicit, in the case of structured control statements. Errors may be present on some execution paths that are not traversed, even though all statements on the path are executed as part of other paths.

Therefore, a more stringent criterion, called branch testing, is that all branch paths in the program are to be traversed at least once. More thorough yet is the criterion, called path testing, that all combinations of branch paths in the program are to be traversed at least once. Because the first requirement often is insufficiently revealing and the third often presents prohibitive combinatorial obstacles, much effort has been directed towards satisfying the second criterion, executing every branch path (see Section 5.3.1).

A particular path is executed if the input data satisfies a system of predicates formed from the conditional branch predicates present in the path. A compiler can build a data structure representing the execution paths of the source program, and using this structure, perform symbolic execution to derive a system of predicates that determines the test data that will cause a given path to be traversed. The predicates are called path constraints. Specific test data can be generated by solving the path constraint predicates. The process of deriving the path constraints and solving the predicates is called automatic test data generation (Miller and Melton, 1975).

User interaction may be required during compilation to select program paths to be considered, or an indication of desired paths may be included in comments or compiler pragmas. A more automatic system may attempt to derive a set of test data that is complete, according to some criteria.

Test data generation systems of the type discussed above are the topic of current research and are not widely used in practice. Test data generators of a different kind have been used with COBOL programs. These generate test data from file specifications, not from the internal logic of the program. Since a COBOL compiler has access to the file specification in a COBOL program, it could provide automatic test data generation.

4.5.5 Correctness Proving

Ideally, given exact program requirements and a formal definition of language semantics, the correctness of a source program should be demonstrable in a rigorous, mathematical fashion from an automated analysis of the source text. A correctness proving system verifies that the program produces the desired output over all input values; testing with actual data is unnecessary.

Experimental programs that perform formal verification have been constructed, based upon principles of theorem proving (Deutsch, 1973, Elspas, et. al., 1973, Good, et. al., 1975). Typically, program verifiers require that the program be augmented with user-written assertions, formally specifying the intended preconditions and effects of program routines. The verifier attempts to construct a formal proof that whenever the input assertions are satisfied at the start of a routine, the output assertions will be satisfied after the routine finishes (Floyd, 1967). An attempt may also be made to prove that a routine will terminate, based, for example, upon convergence proofs of arithmetic computations. Often supplemental information must be provided by the user through inclusion of intermediate assertions.

Current program verifiers suffer several difficulties. First, verifiers are large and complicated, demanding considerable processor time and storage resources. Second, constructing input, output, and intermediate assertions is a tedious, error-prone task for a programmer, similar to writing the program a second time. Compounding this problem is the frequent difficulty in formulating input and output assertions that adequately characterize the intended effect of a module. Further, if a program cannot be verified, the verifier provides no information about selecting the source of the inadequacy, which may be the program, the assertions, or the verifier itself. For these reasons formal verification remains largely experimental, not likely to be included in present compiler systems.

5.0 RUN-TIME CAPABILITIES

A run-time capability requires the specification of the program input in order to be used. Interactive debuggers generally fall into this category. In general, there are three classes of run-time capabilities: those that deal with program control characteristics, those that deal with program data characteristics, and those that deal with program quality or performance characteristics. It is often difficult to separate certain types of run-time capabilities from some compile-time ones. Program instrumentation is such a case. The process of generating an instrumented program, whether it be a modified source or an enhanced object module, is certainly compile-time. However, there are some similar tools which provide the same capabilities, yet are interpretive in implementation. These are strictly run-time.

5.1 CONTROL

Capabilities that analyze dynamic control characteristics of programs are concerned with the flow of control within the program between statements. The ability to insert breakpoints is such a feature.

5.1.1 Symbolic Dynamic Debugging

Dynamic debugging is one of the most time consuming activities involved in programming. A symbolic, high level debugging capability provided by a compiler is a major factor in reducing this time and effort.

In order to work at a consistent level of abstraction, a programmer should be able to debug a program dynamically using the same concepts used to write it. That is, when debugging a program, the programmer should be able to access data objects by name and manage control flow in terms of the source program statements. In particular, debugging should not involve target machine addresses, instructions and numeric codes. These are not the elements used to develop the program and it should not be necessary to learn such details to debug it. Machine level debugging reintroduces the low level concepts the source language was presumably designed to avoid.

Interpreted languages are generally more successful with providing high level debugging than compilers. But high level debugging need not be sacrificed when a language is compiled. A translator can produce information which, in conjunction with a run-time debugging environment, allows accessing data objects by name and performing control flow management by means of source program features, such as statement number and routine name. The basis of this information shared between the compiler and the run-time debugging environment is the symbol table built by the compiler.

The run-time debugging environment minimally should provide capabilities for examining and modifying variables, setting and removing breakpoints, tracing, and single statement execution. A sophisticated dynamic debugger might allow dynamic program correction. With such a system, execution may be suspended to edit the source for a routine, the modified procedure retranslated and linked with the suspended program (as long as its externally visible characteristics have not changed), and execution of the suspended program, now containing the modified routine, resumed.

A symbolic dynamic debugger is a major implementation expense in a compiler system. A further complication in the provision of a dynamic debugger is that many machines do not provide suitable primitive hardware operations to support requirements such as the setting of breakpoints in machine code. Often, hardware limitations must be overcome by devising awkward approximations to the needed functions. However, the reductions in the time and expense of software testing and debugging are well worth the significant effort involved in providing a symbolic dynamic debugger.

5.2 DATA

Capabilities that analyze dynamic data value characteristics of programs are concerned with the storing and retrieval of values into and from program variables and constants. The ability to inspect variables during execution is such a feature.

5.2.1 Symbolic Dynamic Debugging

As previously stated, symbolic dynamic debugging is an important capability of any compiler system. One of the capabilities provided by a symbolic debugger is the inspection of variable values at run-time. The value should be accessible by means of the name used in the high level language, rather than by some other method.

5.2.2 Range Checking

As mentioned previously, range checking of variable values in many cases must be deferred until the program is run. A range-checking compiler often must produce target language statements which perform the check dynamically.

5.2.3 Assertion Checking

Checking of user-embedded assertions often must be deferred until run-time. As is the case with range checking, assertion checking often requires that the compiler produce target language statements that perform the check when the program is run with data.

For example, assuming a compiler that checks a boolean condition following the keyword ASSERT, the following assertion cannot be verified at compile-time, requiring code to be generated to check the assertion:

```
read(index);

ASSERT index > 0;
```

When an assertion condition is evaluated and found false at run-time, the run-time environment can issue a diagnostic, raise an exception condition, or abort execution of the program, depending on user options, language capabilities, and the system environment.

5.2.4 Pointer Checking

Run-time pointer checking is applicable to programming languages that provide facilities for explicit creation and deletion of variables while the program is executing. A variable allocated during execution is called a dynamic variable. Dynamic variables are usually accessed by means of other variables, called pointers, which contain the location of the dynamic object.

The programmer allocates space for a dynamic object by calling a standard language routine to allocate space from a pool, called the heap, set aside for dynamic objects. The allocation routine returns a pointer value, which is the location of the space allocated for the dynamic object. Similarly, the space occupied by a dynamic object can be returned to the pool by calling a deallocation routine, which accepts a pointer variable as a parameter and deallocates the space at that location. Since assignments to pointer variables are usually allowed, a dynamic object can be accessible through more than one pointer variable. An object accessible through more than one pointer variable has aliases.

Pointer checking is necessary because of the problems created by aliases, specifically the dangling pointer problem. A dynamic object can be deleted by passing one of its pointers to the deallocation routine, even though other, ostensibly valid pointers continue to point to the same (now deallocated) object. These pointers are said to be "dangling", i.e., they no longer refer to a valid data object, a fact which cannot be readily determined. Subsequent references to the deallocated object through these pointers are not likely to obtain the correct values, since the space formerly occupied by the object may have been allocated to another object.

The compiler and its run-time routines can provide checking to ensure that this situation does not occur. A simple method is to maintain a count of the number of pointer variables that refer to an object. A call to the allocation routine sets the reference count for the allocated object to one. Upon assignment to a pointer variable, the reference count of the object accessed by the left hand pointer is reduced by one, and the reference count of the object accessed by the right hand pointer is increased by one. An attempt to deallocate a dynamic object has no effect until the count of pointer variables referring to the object is one. Also, the reference count may be useful if the heap space is depleted because the programmer has neglected to deallocate unneeded heap objects. If heap space is exhausted, the run-time environment can examine the reference counts of heap objects, returning to the available pool the space occupied by objects having reference counts of zero.

5.2.5 File Checking

Usually, appropriate data files are a prerequisite to correct program execution. If the data formats of the external file and the internal file specification do not agree, the program is unlikely to perform the expected function.

A compiler can provide some security against this kind of error by checking that external files are consistent with their internal specification. This check is most appropriately performed when the file is opened at run-time. If the data formats do not agree, the run-time environment raises an exception, aborts the program, or prints a diagnostic, depending on the environment.

The compiler can provide this capability by deriving during compilation an encoding of the data format, possibly based upon names or sizes of types in the source program. This encoding is stored with the file when the file is written. When later opened for processing, the encoding of the data format in the program opening the file and the encoding stored with the external file are checked for consistency at run-time.

5.3 STRUCTURAL TESTING AND PERFORMANCE

Many of the above tool capabilities are concerned with refining the program into a form which implements or computes exactly the function that the problem statement requires. However, many tools implement capabilities which are concerned with how well the function is implemented. Such features include dynamic statement counts, and CPU timers.

5.3.1 Statement, Branch, And Path Testing

Common goals of software testing are the execution of every statement, branch, or path in the program at least once. To accomplish these goals, some method must be available for monitoring execution at run-time. Usually, counters in each program block are sufficient to indicate the statements executed during a test run. A program block is a section with the property that if one statement in the section is executed, all statements in the section necessarily are executed. Statements added to collect execution profiles are called software probes.

A compiler can instrument target language programs with instructions to perform execution counting. Instrumenting a program means to insert target language statements, not explicitly coded by the programmer, which will perform the desired activity when the program is run. Enabling the instrumenting option causes a compiler with this capability to implicitly declare an array that will hold the counter values, one for each program block, during execution. The compiler may produce target language statements to increment the counter corresponding to a program section or it may produce calls to a run-time routine to do the incrementing. The run-time environment or a special postprocessor may be used to format and report the information in the counter array after the program finishes.

5.3.2 Performance

In addition to helping satisfy testing criteria, the data gathered from the instrumented source code can be used to suggest sections where code improvements will be most fruitful. The execution data can show the paths of a program that are most frequently traversed in typical situations. If efficiency is an important consideration, these sections should be the first focus of improvement efforts.

6.0 A HIERARCHY OF TOOL CAPABILITIES

The capabilities that have been discussed vary greatly in the amount of information and assistance they provide, in the amount of machine resources they require, and in the degree of invested effort needed to implement them. Capabilities may be grouped according to qualitative considerations of their respective costs and benefits. Three classes of capabilities are distinguished: (1) primary capabilities, which are central to program development support, (2) secondary capabilities, which are useful, but possibly too complex or specialized to be needed in all circumstances or for all languages, and (3) tertiary capabilities, which are helpful, but introduce major implementation difficulties or execution inefficiencies. Capabilities are classified according to informal criteria based upon practice and experience, rather than upon some formal criteria, because of the difficulties in assigning quantifiable figures of merit.

6.1 Primary Capabilities

Some compiler-based capabilities are useful enough and needed often enough to be considered necessary tools for program development. For this reason they are regarded as primary capabilities. In general, they add little complexity to the compiler when compared with their benefits. The following capabilities are placed in this class: library management, cross referencing, program auditing, type analysis, range checking, pointer checking, file checking, dynamic debugging, and interface analysis.

Library management should be considered a primary capability for two major reasons: first, it provides the ability to organize a large software system in a manner understandable to both humans and the compiler. The utility of being able to determine, for example, the modules of a system that reference particular blocks of common data is extremely valuable. Second, a compiler system incorporating library management has available a data base which makes implementation of other primary capabilities, notably global cross referencing, type analysis for procedures, and interface analysis, much easier.

Cross referencing is placed in this class because of its great utility in analyzing the effects of changes to a portion of a program on other portions of the system. In particular, global cross-references, which document the entities shared among more than one module, can significantly reduce the amount of work the programmer needs to do to determine the scope of effect of a change to a data structure. Global cross-references are made much easier by the existence of a library structure that may be referenced by the compiler, or by any other compiler structure which allows interface analysis and general inter-module analysis. Cross-references are also an invaluable capability if present in a compiler for a language which allows implicit declaration, to find occurrences of misspelled variable names.

Type analysis is often the single most valuable capability available to the programmer. Many languages require type analysis as part of any compiler for that language; others do not. For example, a typical error in writing a FORTRAN program is to forget to declare an integer variable having a mnemonic name that unfortunately does not begin with one of the letters I through N. This problem seems to be particularly troublesome in the case of functions. The availability of compiler output which informed the programmer of implicit type conversions at compile-time would eliminate the debugging that is otherwise necessary to remove errors of this type. This analysis must be performed by the semantic phase of the compiler in any case in order to generate correct code; there is no reason why the information cannot be made available to the programmer.

Range, pointer, and file checking and interface analysis can greatly reduce the amount of debugging effort necessary for a given program. In particular, the dangling pointer problem and interface inconsistencies may require an analysis at debugging time of several modules, a nontrivial conceptual problem for a programmer.

The beneficial effects of a good dynamic debugger are so numerous and important that they far outweigh the significant implementation costs of such a debugger. The ability, for example, to examine how a particular memory location changes during the program run rather than only being able to examine its final value in a program dump gives the user much more insight into dynamic program behavior. This capability requires a significant amount of work in a compiler implementation, since a symbol table format useful to the debugger must be defined, the linker and the debugger both must be designed to read it, and the debugger itself must be written.

6.2 Secondary Capabilities

Some useful capabilities can complicate a compiler significantly in regard either to implementation effort, to efficiency of the generated target program, or to efficiency of the translator. Other useful capabilities, though not overly complicated, are specialized or infrequently needed. Capabilities with one or more of these characteristics are considered secondary. Secondary capabilities include: assertion checking, program formatting, program constructing, error correction, macro processing, program structure checking, report generation, flowchart generation, profiling, program restructuring, complexity measurement, reference analysis, units analysis, branch testing, and COBOL test data generation.

Assertion checking, program formatting, report generation, flowchart generation, profiling, program restructuring, complexity measurement, reference and units analysis, and branch testing were placed in this class because they are specialized tools that may be useful over a wide range of applications but do not provide the same benefit to the user as those tools that were placed in the primary class. None of them is particularly difficult to implement; none of them impacts the generated code efficiency to a significant extent, with the possible exception of branch testing.

Program construction is classified as secondary because its efficiency, flexibility, and ease of use are not fully established, since few systems currently implement it. However, the potential benefits, particularly the elimination of syntax errors, are large, and the implementation difficulty is modest compared with tertiary capabilities. For these reasons, it is classified as secondary rather than tertiary.

Error correction is a capability which may often be useful; however, it has two characteristics which cause it to be called a secondary capability. The first is that it does not save significant amounts of program development time. A compiler which gives good diagnostics during syntax analysis can accomplish the same thing, except that the user has to manually insert the change to the source program. Second, the error correction capabilities which currently exist, and probably all those which will ever exist, do not and cannot hope to alter all syntactically incorrect code to look like that which the programmer intended, either because the syntax is extremely far from that needed or because the programmer´s intentions were unclear even to himself. Thus, programmer intervention is necessary to visually validate the changes in any case. Such a tool, when implemented, is part of the syntax analysis phase, and will affect the data structures used internally by that phase both at compiler design time and at compiler runtime.

A macro capability is in this class because a general implementation of such a capability may significantly impact the compiler implementation. In addition, it may also affect the efficiency of the generated code if the users of such a system are insufficiently experienced to differentiate between instances where macros are appropriate and instances where subroutines are appropriate, perhaps leading to unnecessary code duplication.

The utility of structure checking depends largely on the characteristics of the language being compiled. Structure checking can be invaluable in a language having primarily unstructured control flow constructs. However, it is rarely useful in languages having adequate structured control flow statements. Because its utility varies widely, structure checking is classified as secondary.

COBOL test data generation is useful for debugging COBOL programs, especially considering that most COBOL programs perform mainly file processing. This capability is classified as secondary because it is not as difficult to implement as test data generation that is based upon the internal logic of the program.

6.3 Tertiary Capabilities

The distinguishing feature of tertiary capabilities is that they are currently implemented only by research systems that are complex and demand significant user interaction and machine resources. The following capabilities are considered tertiary: symbolic execution, test data generation, and correctness proving.

Although these capabilities may in the future provide the most satisfactory solution to the problem of software reliability, they cannot be readily incorporated into production compilers at present. Implementing these capabilities alone, separately from a compiler, requires a degree of effort that matches or exceeds the effort necessary for implementing most common compilers.

Since tertiary capabilities are likely not to be found in present compiler systems, it may seem a violation of common usage of the term "survey" to include tertiary capabilities in this paper. They have been included as possible compiler-based capabilities for two reasons: First, they operate upon the source program text and are concerned with its correctness and completeness in solving a given problem, thus overlapping the domain and goals of compilers. Second, future technological advances may be anticipated that will increase the feasibility of incorporating these capabilities into compiler-based support systems.

All three of these capabilities are connected, in that they all involve a statement-level walkthrough of the program, with some degree of interpretation or symbolic execution being performed. In the case of symbolic execution, the interpretation uses symbols instead of values. In the case of test data generation, the interpretation uses ranges of input data to differentiate test data into classes. (As mentioned in Section 4.5.4, a different form of test data generation is used with

COBOL programs. This capability is much simpler to implement than generating test data from internal program logic and is therefore classified as secondary.) Finally, correctness proving uses formal assertions as the entities which are carried and interpreted from statement to statement.

7.0 LANGUAGES

The previous sections have focused on general tool capabilities that a compiler can provide, without emphasis on particular languages. Languages have different weaknesses and insecurities, making the useful capabilities depend to a large extent on the language. The following sections will examine FORTRAN, Pascal, COBOL, and BASIC to illustrate the fitting of general capabilities to counteract weaknesses of specific languages.

7.1 FORTRAN

As may be expected of one of the first high level languages, FORTRAN has significant deficiencies. The most recent standard, FORTRAN 77, clarified and improved many undesirable language characteristics, but many shortcomings remain. Enhanced compiler capabilities can make FORTRAN more tractable as a development language. (Except where otherwise noted, the following discussion assumes as its subject FORTRAN 77.)

One of the features of FORTRAN that causes and hides errors is implicit variable declaration. For example, rather than causing a compile-time error, a misspelled identifier has the effect of declaring a new variable. Another, more striking, example is the mistyping of the comma as a period in a DO statement, as in the following:

```
     do 10 x= 1. 100
```

which is a perfectly legal assignment statement. Implicit variable declaration, interacting with the insignificance of blanks in FORTRAN, results in an assignment of the value 1.100 to the variable do10x, instead of a loop. Assuming the convention that variables should be explicitly declared, a compiler with code auditing capabilities can detect misspelled identifiers, diagnosing them as undeclared variables. Reference analysis performed by the compiler will diagnose misspelled identifiers as uninitialized or unused variables. A less helpful tool in this situation is a lexical cross reference listing, which may be inspected manually for identifiers differing only slightly, suggesting a misspelling.

Another error-prone characteristic, related to implicit declarations, is FORTRAN's implicit typing rule, whereby the first letter of an implicitly declared identifier determines its type. If not explicitly declared, variables intended to be real may be implicitly integer, and vice versa. Unintended type assignments can result in loss of precision and unexpected program behavior when arithmetic and relational operations involve mixed modes. Again, a compiler which checks for explicit declarations of variables can help reduce this problem. Also, type analysis performed by the compiler can alert the programmer to ill-advised type coercions in mixed mode operations. Including type information in cross references of variables is helpful for manual detection of unanticipated implicit typing.

FORTRAN lacks many of the structured control statements that are commonly useful in programming. A macro expansion capability that permits the definition and use of structured control statements can greatly simplify FORTRAN programming, obviating use of labels and GOTO statements. Code restructuring is another useful, although more difficult to implement, defense against this inadequacy.

Compilation of FORTRAN subprograms is independent, i.e., no checking of interface consistency between modules is performed. Accordingly, interface errors are common. Most FORTRAN compilers do not detect parameter type mismatches, incorrect number of parameters, or COMMON variable misalignment or omission. These errors should be detected by automated tools, rather than requiring tedious manual inspections. The compiler can check interfaces without introducing inordinate complexity by maintaining interface information on the host file system. Interface summaries produced by a compiler are much less helpful and secure.

The problem of FORTRAN interfaces can be relieved to a large extent by a source library management facility. The many routines of a large program usually share data by means of numerous COMMON blocks. The probability of error is very high if the specification for a busy COMMON block changes. It is exceedingly tedious to trace through all the routines which use the COMMON block, changing the specification. A library management capability can provide automatic updating of routines which use a modified COMMON block. Library management can also support the capability to perform interface analysis by maintaining the information needed for interface checking in a member of the library.

FORTRAN contains a number of features through which the compiler implementation can influence program behavior. These features should be avoided, and can be detected by a compiler with the capability to do program auditing. Some practices relying on implementation dependencies can be checked at compile-time. An example is the passing of a literal constant as a parameter to a subprogram. If an assignment is made to the parameter by the subprogram, the value of the literal constant may be changed, making incorrect subsequent computations involving the constant. A code auditing compiler can warn of a potential error when literal constants appear in parameter lists. Depending on local variables to retain their values between successive calls of a subprogram is another example of unsafe programming. This practice assumes a particular strategy for variable allocation and should be avoided. A compiler with the capability to do reference analysis can detect uses of local variables which suggest that values from previous calls are being assumed valid.

Other implementation dependent characteristics must be checked at run-time. For example, before the most recent FORTRAN standard, the action of a computed GOTO statement was left undefined for the case that the integer expression was nonpositive or was larger than any alternative. Similarly, if the initial value of the index variable of a DO loop was less than the exit value, assuming a positive step, or was greater than the exit value, assuming a negative step, the behavior of the DO loop was implementation dependent. A range checking FORTRAN compiler can insert target language statements to verify that the values at run-time are allowable. If the system environment permits user exception handling, a graceful recovery from an error may be accomplished. In a less favorable environment, a diagnostic may be issued. In either case, rather than relying on a specific implementation for correctness, the program should be diagnosed as erroneous.

In addition to reducing implementation dependencies, range checking can provide security in array references, a frequent source of errors. If an array is declared in a program unit, its bounds are available so that indices in array references can be checked, either at compile-time or, if necessary, at run-time. Arrays that are parameters to subprograms require more sophistication from the compiler, since the dimensions of the array parameter can vary from one subprogram call to the next. The array bounds are not known when the subprogram is compiled. The capability to check array indices in this case is simplified if the compiler also performs interface analysis. The bounds of the array passed as a parameter to the subprogram are part of the interface information that must be maintained for modules calling the subprogram. The compiler can use this information to perform run-time checks of the array indices.

Because FORTRAN is not strongly typed, subtle errors can result from implicit type coercions. Many compilers permit indiscriminant mixes of operand types in arithmetic and relational operations, the result possibly depending on target language data representations adopted by the implementation. Combining real operands with other types can result in loss of precision. Type analysis performed by the compiler can warn of error-prone type coercions and type inconsistencies, helping to eliminate type errors.

7.2 Pascal

Pascal is a milestone in programming language design. Because of the language definition, capabilities that are enhancements to compilers for other languages are required and routine parts of Pascal compilers. Nevertheless, Pascal contains characteristics that tend to be associated with errors, some of which may be avoided given a suitably supportive compiler.

A major problem with using Pascal in large software systems is that implementations often do not provide a facility to separately compile procedures. A minor change anywhere in a program forces the entire program to be recompiled, a significant expense with large systems. Contrary to the design approach of the language, many implementations allow independent compilation of procedures, performing no checks of interface consistency.

Pascal is enhanced considerably as a systems implementation language if procedures can be compiled separately, maintaining interface security. As suggested previously, the compiler can produce the information it needs to check interfaces and save it on the host file system. When beginning compilation of a separate procedure, the interface information is made available to the compiler so that appropriate checks may be performed. If separate compilation is integrated within a library management facility, the management of interface information and compilation status can be automated.

Semantic auditing of Pascal programs can detect a number of common errors. A frequent error results from misuse of parameter modes in procedures. If a variable in a procedure call is to be changed by the procedure, the corresponding formal parameter in the procedure heading must be preceded by the keyword VAR. A common oversight is to omit the VAR preceding a variable parameter, resulting in the failure to change the variable in the calling environment. The change is made to the local parameter of the procedure, but not to the actual variable in the calling environment. Thus, a procedure that otherwise may be correct does not update the value of one of its parameters, often an elusive error. Assigning a value to a parameter not preceded by VAR is either poor programming practice or an inadvertant omission of the VAR. It is helpful if a compiler notes this flaw. CASE statements may be audited to ensure that the selector variable must have the value of one of the selector constants (see Section 4.4.6).

Coding standards can restrict poor programming practices allowed by Pascal. Pascal control structures are rich enough to make the use of GOTO statements seldom, if ever, justifiable. In particular, GOTO statements that jump out of procedures are troublesome, both for the compiler and for readers of the program. A compiler can enforce organizational programming standards forbidding such usage.

The benefits of strong typing can be compromised if the compiler provides no checking of the types of external files (see Section 5.2.5). Many Pascal compilers perform no checking of types when a file is opened at run-time. GET operations from files in such cases may retrieve data having a type that is inconsistent with the file type as given in the source program. A simple method of file checking is to derive an encoding, or key, based upon the data type in the file. This key is checked for consistency at run-time when a file is opened.

A troublesome insecurity of Pascal is its definition of discriminated unions, called variant records. A discriminated union is a record which may contain different components, depending on the value of the discriminant, or tag, variable contained in the record. The tag determines the names, number, and types of components that the record contains. Pascal discriminated unions are insecure because the programmer is responsible for setting the tag value before assigning values to variant fields and for checking the tag before accessing variant fields. If the programmer omits setting the tag variable or sets it incorrectly, the value of the tag can become inconsistent with the values contained in the variant fields, reintroducing the insecurities eliminated by strong typing. Similarly, if the programmer does not check the tag variable, the data values contained in the record may be inconsistent with the variants referenced, resulting in inappropriate operations.

Several capabilities are helpful in reducing or eliminating errors involving discriminated unions. The most primitive assistance is a cross reference of statements that change tag variables and fields of variant records. More helpful is a run-time check that the tag value is consistent with accesses to variant fields. Of course, the most helpful capabilities detect errors at compile-time. Semantic auditing can ensure that fields are accessed only within program sections that have first checked the tag value for consistency. Alternatively, the compiler can verify that accesses to variant fields are consistent through data flow considerations at compile-time. Since the validity of variant references is sometimes indeterminant at compile-time, a data flow capability should be augmented by run-time checking where necessary. (Although their use is discouraged, free unions, i.e., variant records containing no tag field, are allowed in Pascal. A compiler can perform no consistency checks on uses of variants of a free union.)

Many implementations of Pascal's dynamic variables make the programmer susceptible to the dangling pointer problem (see Section 5.2.4). Storage for a dynamic variable in Pascal is allocated by a call to the routine NEW, which takes a pointer variable as a parameter and assigns to it the address of the space allocated. When the space is no longer required, it may be returned to the pool of available space by a call to the function DISPOSE, which takes a pointer variable as a parameter and deallocates the space at that address. The dangling pointer problem occurs when a pointer to a dynamic object is passed to DISPOSE, even though other pointer variables continue to contain the location of the deallocated object. Since the space occupied by the deallocated object is in the available pool, it may be allocated to another object, possibly of a completely different type than the original object. References via the dangling pointers can thus access values of an object of a completely different type than those intended. A compiler performing run-time pointer checking will ensure that an object is deallocated only when it is no longer accessible through any variable. The technique mentioned above of keeping reference counts for dynamic objects is one method of eliminating dangling pointer problems.

7.3 COBOL

COBOL is a language designed in the early 1960s for business applications. It is the most widely used computer language, but few of its design concepts have had significant influence on later languages. Both of these facts may be partially attributed to its orientation toward business data processing, a major area of computer application but one in which the problems are of a somewhat unique character: relatively simple algorithms coupled with high volume input-output. Most other languages have been designed to implement relatively complex algorithms with input-output a secondary consideration. In addition to the goal of handling large scale input-output, other important design criteria for COBOL were source program transportability and highly readable, self-documenting, English-like source text.

These design goals shaped both the strengths and weaknesses of the language, although some of the weaknesses result from the lack of experience with computer language design at the time when COBOL was specified. The major strengths of the language are transportability of source code, good file handling capability, and capability for specification of complex, hierarchical data structures useful for the type of file processing common in business applications. The major weaknesses are inelegant structured control constructs, difficulty in producing efficient object code because of complicated and machine dependent data representation, verbose, cumbersome syntax, and lack of good partitioning mechanism for data or for code. The adverse effects of these weaknesses can be reduced through the use of compiler-based tools, procedures for controlling the production of programs, and procedures for analyzing the code produced to minimize inefficient or error-prone constructs.

One of the keys to transportability of COBOL programs is the character string representation of data. Data descriptions can be almost entirely independent of particular hardware characteristics, such as word length or number representation. This default data representation is adopted unless the data item is declared with the clause USAGE IS COMPUTATIONAL. While this representation enhances program transportability, it makes the problem of efficient code generation difficult. Also, it encourages the use of implied mixed mode arithmetic on a complex range of types, further degrading object code efficiency and leading possibly to unexpected results. This problem can be alleviated by a semantic cross referencer and a semantic auditor to flag or prohibit implied mixed mode arithmetic where the result will be of questionable reliability or where extremely inefficient code will be generated.

Due to the static nature of COBOL storage allocation, all data is essentially global. Modern programming methods have emphasized the practice of localizing data definition and usage to increase both reliability and understandability. Following this practice is difficult in COBOL, but a detailed lexical cross reference can compensate somewhat for this language deficiency by providing easier access to all usages of data elements within a program.

The COPY facility for including source text from a library into a program improves the ease of writing COBOL. To use this facility effectively, strict configuration control must be exercised to prevent copying incorrect or inappropriate modules. A library management facility integrated within a COBOL compiler system could provide this and other configuration management functions, without undue restrictions imposed by the host system.

The overly permissive semantic rules of COBOL have given rise to a number of articles suggesting rules and restrictions for writing structured COBOL programs (Van Gelder, 1977, McLure, 1975). The use of lexical, syntactic, and semantic code auditors would be useful for enforcing coding guidelines, and reducing programmer susceptibility to common COBOL errors. Some of the rules that have been suggested are:

1. Execution flow cannot cause control to fall or jump into a SECTION.

2. All IF statements will be required to have an ELSE clause to serve as a marker to indicate the boundary of the IF statement, since COBOL syntax provides no block structure.

3. The maximum level of nesting for IF statments will be three.

4. Each SECTION will be followed immediately by its EXIT

SECTION. The EXIT SECTION will be named EXIT-<name> where <name> is the SECTION procedure name. The only statement which may appear in the EXIT SECTION is the EXIT statement.

5. The values of any index-names or identifiers used as control variables in the PERFORM statement should not be modified in the SECTION being performed.

6. A GOTO statement can only be used to jump into another part of the same SECTION in which it appears.

Because COBOL syntax is often verbose and repetitive, macro processors have become useful and popular adjuncts to COBOL compilers (Triance and Yow, 1980). Some of these have been incorporated into compilers, while others are used as preprocessors. Macro processors not only help in abbreviating repetitive syntax, they also can provide more natural structured control constructs and other language features that may not be available in particular versions of COBOL.

Many existing COBOL programs are the result of a large investment in time and manpower. Often they perform essential and uninterruptable processing for organizations. Unfortunately, many of these programs are written without adherence to rules for making them easy to understand and maintain. Maintenance of such programs can be costly. The ability to restructure such programs according to guidelines for maintainable software would be valuable in reducing maintenance costs without requiring a major reprogramming effort.

7.4 BASIC

Originally intended as a simple programming language for computer novices, BASIC does not contain many of the features which, although they lead to the creation of more reliable, readable, and maintainable programs, also make the programming language itself more difficult to assimilate. Thus, there is room for many valuable capabilities in most BASIC systems.

Many BASIC systems are interpretive, influencing significantly the capabilities that are useful or implementable. For example, structure analysis is more difficult in an interpretive environment. Regarding other capabilities, an interpreter has advantages over a compiler, for example in providing symbolic dynamic debugging. An interpretive system has available at "run-time" all of the structure of the original program; thus, it can easily provide for user interaction at a level close to that of the source. Also, an interpreter often can provide more extensive capabilities, for example a feature whereby the user may execute selected BASIC statements when the program is stopped at a particular point, perhaps to print the values of intermediate results or to change the value of a variable.

Although modern implementations of BASIC often support an extremely large, general-purpose language useful for both scientific and business applications, it is important that these implementations retain a language subset that is learned easily by the novice user, the original intent of BASIC. Thus, tools that increase the difficulty of learning the system probably should be avoided. These might include, for example, library management (although most systems do not allow separately compiled modules in any case), program constructing, and macro systems. Nonetheless, several useful capabilities significantly assist all programmers, including novices.

One of the most dangerous features of BASIC is the GOSUB construct, which allows the programmer to declare a local procedure and invoke it without passing arguments. The procedure is not delineated by any syntax; therefore, it is possible to drop from main-line code into a GOSUB procedure. Rarely is this sequence intentional; a compiler-based system can perform flow analysis to determine if this control sequence can occur, issuing an appropriate warning in such cases.

Another unique characteristic of BASIC is that scalars and arrays may have the same name, although the data objects are unrelated. This feature, in conjunction with the fact that vectors whose dimension is less than ten need not be declared, can lead to errors if the user inadvertently adds or omits a vector subscript. A cross-reference facility, and perhaps a warning message when a scalar and an array have the same name, can significantly reduce the debugging time required to detect such errors.

Most BASIC systems, interpretive or not, do not check for or detect interface errors. In fact, some systems allow the number of parameters on the calling and receiving sides to disagree, an obviously dangerous practice. Regardless of whether the implementation approach is compilation or interpretation, the system can easily check the number and types of parameters, at call time for an interpretive system, and at compile time for a compiler (assuming that independently compiled modules are not supported).

BIBLIOGRAPHY

Boyer, R. S., B. Elspas, and K. N. Levitt, "SELECT: A formal system for testing and debugging programs by symbolic executions," Proc. 1975 International Conf. on Reliable Software, Apr. 1975.

Clarke, L., "Generating test data and symbolically executing programs written in ANSI FORTRAN," IEEE Trans. Software Engineering, Sep. 1976, pp. 215-222.

Deutsch, L. P., "An interactive program verifier," PhD Thesis, UC/Berkeley, California, 1973.

Elspas, B., K. N. Levitt, and R. J. Waldinger, "An interactive system for the verification of computer programs," SRI Project 1891, Stanford Research Institute, Menlo Park, California, 1973.

Fischer, C. N., D. R. Milton, and S. B. Quiring, "An efficient insertion-only error-corrector for LL(k) parsers," Proc. 4th ACM Symposium on Principles of Programming Languages, 1977, pp. 97-103.

Floyd, R. W., "Assigning meanings to programs," in Proc. Symposium Applied Mathematics, Vol. 19, American Mathematical Society, Providence, R. I., 1967, pp. 19-32.

Good, D. I., R.L. London, and W. W. Bledsoe, "An interactive program verification system," Proc. International Conf. on Reliable Software, Apr. 1975.

Habermann, A. N., "The Gandalf research project," Computer Science Research Review, Carnegie-Mellon University, 1980, pp. 28-35.

Halstead, M. H., Elements of software Science, New York: Elsevier, 1977.

Howden, W. E., "Symbolic testing and the DISSECT symbolic evaluation system," IEEE Trans. Software Engineering, Vol. SE-3, No. 4, Jul. 1977, pp. 266-278.

Huet,G., G. Kahn, and P. Maurice, "Environnement de programmation Pascal," Technical report, IRIA Rocquencourt, Nov. 1977.

King, J., "A new approach to program testing," Proc. 1975 Int. Conf. on Reliable Software, Apr. 1975.

King, J., "Symbolic execution and program testing," CACM, Jul. 1976.

McCabe, J. T., "A Complexity measure," IEEE Trans. Software Engineering, Vol SE-2, No. 4, Dec. 1976, pp. 308-320.

McClure, Carma L., "Structured Programming in COBOL," SIGPLAN Notices, April 1975. Vol. 10, No. 4, pp.25-33.

Osterweil, L. J., and L. D. Fosdick, "DAVE - a validation error detection and documentation system for FORTRAN programs," Software Practice and Experience, Vol. 6, 1976, pp. 473-486.

Teitelbaum, Ray T., "The Cornell program synthesizer: A microcomputer implementation of PL/CS," Technical report, Department of Computer Science, Cornell University, 1979.

Triance, J.M. and Yow, J.F.S., "MCOBOL-A Prototype Macro Facility for COBOL." CACM August, 1980, Vol. 23, No. 8, pp.432-439.

Van Gelder, Alan, "Structured Programming in COBOL: An Approach for Application Programmers," CACM Jan. 1977, Vol 20, No. 1, pp.2-12.

NBS-114A (REV. 2-80)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET (See instructions)	1. PUBLICATION OR REPORT NO. 81-2423	2. Performing Organ. Report No.	3. Publication Date

4. TITLE AND SUBTITLE

Compiler-Based Programming Support Capabilities

5. AUTHOR(S)

Gary Bray, Roger Lipsett, William Bail, and Victor Berman

6. PERFORMING ORGANIZATION (If joint or other than NBS, see instructions)

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

Intermetrics, Inc.
4733 Bethesda Avenue
Bethesda, Maryland 20014

7. Contract/Grant No.
NB79SBCA0131

8. Type of Report & Period Covered
Final Report

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (Street, City, State, ZIP)

National Bureau of Standards
Department of Commerce
Washington, D. C. 20234

10. SUPPLEMENTARY NOTES

□ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT (A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)

An effort to determine a set of features offered by program analysis and testing tools that could be feasibly implemented in a compiler is reported. Currently, program analysis and testing tools offer features that require syntactical analysis of a program in a manner similar to compilers. Much of the information that is generated during compilation could be used to aid program development in other ways. It was the goal of this effort to identify a set of software tool features and develop a methodology for combining these into a compiler.

12. KEY WORDS (Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)

Compilers; Dynamic Analysis; Programming Aids; Software Development; Software Engineering; Software Tools; Static Analysis

13. AVAILABILITY

☒ Unlimited

□ For Official Distribution. Do Not Release to NTIS

□ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

☒ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES
67 pages

15. Price
8.00